やさしいC++ 第2版

高橋 麻奈 著

Mana Takahashi

SOFTBANK Publishing

やさしいC++ 第2版

高橋 麻奈 著

Mana Takahashi

SOFT BANK Publishing

まえがき

現在、C++言語を扱うさまざまな開発ツールが発売されています。プログラムづくりはやってみたいけれども、「C++はちょっとハードルが高いかな…」と思っていらっしゃる方も多いのではないでしょうか。

本書は、そんな方々のためのC++言語の入門書です。プログラミングを学んだことがない方でも、無理なく学習できるように構成されています。プログラミングの初歩から解説しているので、C言語など、ほかのプログラミング言語の知識は必要ありません。また、イラストを豊富に使って、できるだけわかりやすく概念を図解するように心がけました。

本書には多くのサンプルプログラムが掲載されています。プログラミング上達への近道は、実際にプログラムを入力し、実行してみることです。ひとつずつたしかめながら、一歩一歩学習を進めていってください。なお、サンプルプログラムは、Microsoft Visual C++、GNU C++で検証しています。

本書が皆さまのお役にたつことを願っております。

著者

C++言語開発環境の使いかた

C++のプログラムは、本書の第1章で説明するように、①ソースコードの作成→②コンパイルの実行（→③リンクの実行）→④プログラムの実行、という順番で作成します。そこで、ここではC++言語プログラムを開発する代表的なツールである「Microsoft Visual C++」の使いかたを通して、プログラム実行までの手順を説明しておきましょう。①〜④のくわしい意味については、第1章を参照してください。

Microsoft Visual C++の使いかた

Microsoft Visual C++（マイクロソフト（株））は、パソコンショップなどで販売されている統合開発環境です。

使用前の設定

Microsoft Visual C++の説明書を参考に、インストール・起動してください。

プログラムの作成手順

1. メニューから、［ファイル］→［新規作成］→［プロジェクト］を選択します。［新しいプロジェクト］画面が表示されたら、［Win32 プロジェクト］を選択します。［プロジェクト名］に「Sample1」などと入力し、［場所］にファイルを保存するフォルダ名を指定してください。

新しいプロジェクト
プロジェクトの種類(P):
Visual Basic プロジェクト
Visual C# プロジェクト
Visual C++ プロジェクト
セットアップ/デプロイメント プロジェクト
その他のプロジェクト
Visual Studio ソリューション
テンプレート(T):
Managed C++ クラス ライブラリ
MFC ActiveX コントロール
MFC DLL
MFC アプリケーション
Win32 プロジェクト
メイクファイル プロジェクト
Win32 コンソール アプリケーションかほかの Win32 プロジェクトです。
プロジェクト名(N): Sample1
場所(L): C:¥YCCSample¥01
参照(B)...
プロジェクトは C:¥YCCSample¥01 に作成されます。
詳細(E)
OK
キャンセル
ヘルプ
① 選択します
② 入力します

2. 「Win32アプリケーションウィザード」が表示されます。左端の「アプリケーションの設定」を選択します。アプリケーションの種類を「コンソールアプリーション」とし、[空のプロジェクト]をチェックしてください。最後に[完了]ボタンを押します。

3. [ファイル]→[新しい項目の追加]を選択します。[新しい項目の追加]画面があらわれたら、[C++ファイル]を選択します。[ファイル名]に、「Sample1.cpp」などと入力し、[場所]にSample1プロジェクトを作成したフォルダを指定してください。最後に「開く」ボタンを押すと、ソースファイルが作成されて入力できるようになります。(・・・**①ソースコードの作成**)

4. サンプルを入力したら、メニューから［ビルド］→［ソリューションのビルド］を選択します。ソースファイルの保存・コンパイルが行われます。（・・・**②コンパイルの実行＋③リンクの実行**）

① 本書のとおりに作成します

② ［ビルド］→［ソリューションのビルド］を選択します

5. ［デバッグ］→［デバッグなしで開始］を選択します。コマンドプロンプトが自動的に起動して、プログラムが実行されます。実行を終了するには、どれかキーを押してください。（・・・**④プログラムの実行**）

ほかのサンプルプログラムを作成する場合は、1の手順に戻って、新しいプロジェクトを作成しなおしてください。また、4の手順に戻ってディタ部分に新しいコードを入力することもできます。ただし、4の手順に戻った場合はすでに作成したプログラムに上書きされます。

第10章のようにファイルを分割してコンパイルする場合は、3の手順に戻って、新しい項目を追加します。

Micorosoft Visual C++のくわしい使い方については、ヘルプファイルや参考書をお読みください。

Contents

Lesson 15 クラスに関する高度なトピック …… 461

Lesson 16 ファイルの入出力 …… 505

コラム

Lesson 1

はじめの一歩

この章では、C++ 言語でプログラムを作成する手順について学びます。C++ 言語の勉強をはじめたばかりの頃は、耳慣れないプログラム言葉に苦労することもあるかもしれません。しかし、この章でとりあげるキーワードがわかるようになれば、C++ 言語の理解も楽になるはずです。ひとつずつしっかりと身につけていきましょう。

Check Point!

- プログラム
- C++言語
- 機械語
- ソースファイル
- コンパイル
- オブジェクトファイル
- リンク
- プログラムの実行

1.1 C++のプログラム

プログラムのしくみ

本書を読みはじめている皆さんは、これからC++で「プログラム」を作成しようと考えていることでしょう。私たちは毎日、コンピュータにインストールされたワープロ、表計算ソフトなど、さまざまな「プログラム」を使っています。ワープロのような「プログラム」を使うということは、

文字を表示し、書式を整え、印刷する

といった特定の「仕事」をコンピュータに指示し、処理させていると考えることもできます。

コンピュータは、さまざまな「仕事」を正確に、速く処理できる機械です。「プログラム」は、コンピュータに対してなんらかの「仕事」を指示するためのものです。

私たちはこれから、C++を使って、コンピュータに処理を行わせるための「プログラム」を作成していくことにします。

図 1-1 プログラム
私たちはコンピュータに仕事を指示するために「プログラム」を作成します。

プログラミング言語C++

コンピュータになんらかの「仕事」を処理させるためには、お使いのコンピュータが、その仕事の「内容」を理解できなければなりません。このためには本来、**機械語**（machine code）と呼ばれる言語でプログラムを作成することが必要になります。

しかし困ったことに、この機械語という言語は、「0」と「1」という数字の羅列からできています。コンピュータなら、この数字の羅列（＝機械語）を理解することができるのですが、人間にはとうてい理解できる内容ではありません。

そこで、機械語よりも「人間の言葉に近い水準のプログラム言語」というものが、これまでにいくつも考案されてきました。本書で学ぶ**C++言語**も、このようなプログラミング言語のうちのひとつです。

C++は、**コンパイラ**（compiler）というソフトウェアを使って、機械語に翻訳されることになっています。この機械語のプログラムによって、コンピュータが実際に「仕事」を処理することができるのです。

それではさっそく、C++を学んでいくことにしましょう。

1.2 コードの入力

「コード」とは

C++言語でプログラムを作成するためには、どんな作業が必要となるのでしょうか？　ここでは最も基本的なプログラムの作成方法をみていくことにしましょう。

プログラムを作成する場合には、まず最初に

テキストファイルに、C++言語の文法にしたがってプログラムを入力していく

という作業をはじめます。かんたんなC++のプログラムは、

- Windowsの「メモ帳」
- UNIXの「vi」

といった「テキストエディタ」を使って作成できるようになっています。図1-2は、C++のプログラムをテキストエディタに入力している画面です。本書では、これからこんなふうにテキストエディタにプログラムを入力していくのです。

このテキスト形式のプログラムは、**ソースコード**（source code）と呼ばれています。本書では、このプログラムのことを、単純に**コード**と呼ぶことにします。

```
#include <iostream>
using namespace std;

int main()
{

   cout << "ようこそC++へ!¥n";

   return 0;
}
```

プログラムを作成します

図 1-2　C++ で記述したコード (Windows のメモ帳の場合)
C++ のプログラムを作成するには、テキストエディタでコードを入力することからはじめます。

商用の開発環境を使ってみる

なお、ソースコードを作成するには、店頭で販売されているC++ 言語開発環境(Microsoft Visual C++ など) を使うこともできます。これらの製品には、あらかじめ独自のテキストエディタが組み込まれています。本書冒頭でも、Microsoft Visual C++ について解説しているので参考にしてください。

Sample1 - Microsoft Visual C++ [デザイン] - Sample1.cpp*

ファイル(F) 編集(E) 表示(V) プロジェクト(P) ビルド(B) デバッグ(D) ツール(T) ウィンドウ(W) ヘルプ(H)

Sample1.cpp*

```
#include <iostream>
using namespace std;

int main()
{
   cout << "ようこそC++へ!¥n";

   return 0;
}
```

開発環境のエディタを使うこともあります

図 1-3 C++で記述したコード(Microsoft Visual C++の場合)
商用の開発環境では、独自のエディタが組み込まれている場合があります。

ワープロソフトは使わないこと

テキストエディタと似たソフトに、文字の大きさや太さを設定できる「ワープロソフト」があります。しかし、文字の大きさなどの書式情報を保存するワープロソフトは、C++のコードを保存するには適切ではありません。ワープロソフトはプログラムを作成する際には使わないようにしましょう。

テキストエディタにコードを入力する

それでは、テキストエディタにC++の「コード」を入力してみましょう。次の点に注意しながら文字を入力してみてください。

■ 英数字は全角ではなく、半角で入力してください。

■ 英字の大文字と小文字は異なる文字として区別されています。大文字・小文字はまちがえないように入力してください。たとえば、「main」の文字を「MAIN」としてはいけません。
■ 空白の部分は、スペースキーまたは Tab キーを押してください。
■ 行の最後や、何も書かれていない行では、Enter キーを押して改行してください。このキーは、実行キー、↵ キーなどと呼ばれている場合もあります。
■ セミコロン（;）やカッコの違いに気をつけて入力してください。
■ {　}、[　]、(　)の違いに気をつけて入力してください。
■ 0（ゼロ）とo（英字のオー）、1（数字）とl（英字のエル）もまちがえないで入力してください。

入力が終わったら、最後にファイルに名前をつけて保存します。通常、C++のソースコードのファイル名には、最後に「.cpp」をつけて保存します。これを**拡張子**といいます。つまりファイル名は、「<自分がつけた名前>.cpp」という名前にします。

ここでは、ファイルに「Sample1.cpp」という名前をつけて保存してみましょう。

Sample1.cpp ▶ はじめてのコード

```
#include <iostream>
using namespace std;

int main()
{
   cout << "ようこそC++へ!¥n";

   return 0;
}
```


こうしてできあがった「Sample1.cpp」が、はじめて作成したC++の「コード」です。このコードを保存したファイルは、**ソースファイル**（source file）と呼ばれています。

1.3 プログラムの作成

コンポイルのしくみを知る

1.2節で入力したSample1は、

コンピュータの画面に「ようこそC++へ！」という文字を表示する

という処理をするプログラムです。

せっかく入力したコードです。「早く動かしてみたい！」と思われていることでしょう。

しかし、あせりは禁物です。ソースファイルを作成しただけでは、すぐプログラムを実行して文字を表示させることはできません。C++で記述されたコードは、コンピュータが直接内容を理解して処理できるように、

機械語のコードに変換する

という作業を行う必要があるのです。

C++を機械語にする作業は、**コンパイル**（compile）と呼ばれています。この作業には、**コンパイラ**（compiler）と呼ばれるソフトウェアを使います。

ソースファイル（C++言語）

オブジェクトファイル（機械語）

001
001

コンパイラ

図 1-4 コンパイル
ソースコードを機械語のコードに翻訳する作業をコンパイルといいます。

コンパイラを実行する

コンパイルのしかたは、お使いのC++開発環境によって異なってきます。コンパイル方法については、お使いの開発環境に添付されている説明書をお読みください。なお、本書の冒頭では、Microsoft Visual C++での実行方法について解説しています。

コンパイルを行うと、通常、ソースコードが保存されているフォルダ（ディレクトリ）に、機械語に翻訳されたファイルが新しく作成されます。このファイルのことを、**オブジェクトファイル**（object file）と呼びます。

エラーが表示されてしまったら？

コンパイルをしようとしたところ、画面にエラーが表示されてオブジェクトファイルが作成できない場合があります。こんなときには、入力したコードをみなおして、まちがいがないかどうか確認してください。まちがいをみつけたらその部分を訂正し、ソースファイルを保存してから、もう一度コンパイルしてみましょう。正しくコンパイルできたでしょうか？

C++は、英語や日本語と同じように、「文法」規則をもっています。もし私たちが、C++の文法にしたがわないコードを入力した場合は、コンパイラはそのコードを理解することができません。つまり、コンパイラは、ソースコードを正しく機械語に翻訳できないのです。このとき、コンパイラはエラーを表示して、文法などの誤りを訂正するように指示を与えてくれるようになっています。

オブジェクトファイルをリンクする

さて、C++では、コンパイルを終了したあと、続けて

いくつかのオブジェクトファイルをつなぎあわせて1つのプログラムを作成する

という作業を行います。C++では、作成したオブジェクトファイルに、ほかのプログラムでも共通して使うオブジェクトファイルをつなぎあわせて、実際に実行

できる形式のプログラムを1つ完成させるようになっているのです。

この作業は、**リンク**（link）と呼ばれています。リンクを行うソフトウェアのことを**リンカ**（linker）と呼びます。

本書の冒頭で紹介した開発環境では、コンパイルのあとに自動的に必要なファイルがリンクされるようになっています。

リンクについては、第10章でもくわしく学ぶことにします。

図1-5 リンク
オブジェクトファイルをつなぎあわせて1つのプログラムをつくることをリンクといいます。

1.4 プログラムの実行

プログラムを実行する

それでは、さっそくできあがったプログラムを実行してみましょう。

Windowsの場合、作成されたプログラムには、「Sample1.exe」などという名前がつけられます。

プログラムの実行方法にはさまざまな方法がありますが、最もかんたんな方法は「MS-DOSプロンプト」（コマンドプロンプト）から実行する方法です。

まず、MS-DOSプロンプトを起動します。次に、Sample1.exeが作成されたディレクトリに移動します。ここで、「Sample1」と入力し、Enter キーを押します。

Sample1の実行方法

```
C:¥YCCSample¥01>Sample1 ↵
```

プログラムを実行します

ソースファイルを保存したディレクトリで・・・

プログラムが実行されると、画面上に文字が表示されます。

Sample1の実行画面

```
ようこそC++へ!
```

```
コマンド プロンプト
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:¥>CD YCCSample¥01

C:¥YCCSample¥01>Sample1
ようこそC++へ!

C:¥YCCSample¥01>
```

プログラム名を入力して Enter キーを押すと・・・

プログラムが実行されます

図 1-6　プログラムの実行
プログラムを実行すると、「ようこそC++へ!」という文字が画面に表示されます。

うまく実行できたでしょうか？ なお、このほかにもいろいろな実行のしかたがあります。本書冒頭やお使いの開発環境の説明書を参考にしてみてください。

では最後に、この章で学んだプログラムの作成・実行手順をまとめておきましょう。本書2章以降のサンプルコードも、このような手順にしたがって入力し、実行していくことになります。手順をしっかりと身につけておいてください。

① テキストエディタなどにC++のコードを入力する
↓
② コンパイラを起動して、作成したソースファイルをコンパイルする
↓
③ リンカを起動して、コンパイルされたオブジェクトファイルをリンクする
↓
④ 作成されたプログラムを実行する

1.5 レッスンのまとめ

レッスンのしめくくりとして、この章で学んだことをまとめておきましょう。この章では、次のようなことを学びました。

- プログラムは、コンピュータに特定の「仕事」を与えます。
- C++ のコードは、テキストエディタなどに入力します。
- C++ のコードは、大文字と小文字を区別して入力する必要があります。
- ソースファイルをコンパイルすると、オブジェクトファイルが作成されます。
- オブジェクトファイルをリンクすると、実行できるプログラムが作成されます。
- プログラムを実行すると、指示した「仕事」が行われます。

この章では、C++ のコードを入力し、プログラムを作成する手順を学び、最後に実行してみました。しかしこの章では、入力したC++ コードの内容についてはふれませんでした。次の章からさっそく、C++ のコードの内容について学んでいくことにしましょう。

練習

1. 次の項目について○か×で答えてください。

① C++ のソースコードは、そのままの形式で実行することができる。

② C++ では、英字の大文字と小文字を区別して入力する。

③ C++ では、半角英字と全角英字を区別しないで入力できる。

④ ソースコード中の空白は、必ずスペースキーを押して空白とする。

⑤ C++ のソースコードは文法規則が誤っていた場合でも、常にコンパイルできる。

2. 本文中で作成したSample1.cppを、Sample2.cppという名前で保存しなおしてください。Sample2.exeという名前のプログラムを作成・実行してください。

Lesson 2

C++ の基本

第１章では、C++ のコードを入力し、コンパイラを使ってプログラムを作成する方法を学びました。それでは、これから私たちはどのようなコードを入力していったらよいのでしょうか？　コードを記述し、プログラムを作成するためには、C++ の文法規則を知らなければなりません。この章では、C++ の文法の基本を学ぶことにしましょう。

Check Point!

- 画面への出力
- main()関数
- ブロック
- コメント
- プリプロセッサ
- インクルード
- リテラル
- エスケープシーケンス

2.1 画面への出力

新しいコードを入力する

第1章では、画面に1行表示するプログラムを作成しました。コンピュータに「仕事」を行わせせることができたでしょうか？　この章では、さらに多くの行を表示するプログラムを作成してみましょう。

下のコードをエディタに入力し、保存してください。

Sample1.cpp ▶ 画面に文字列を出力する

```
//画面に文字を出力するコード
#include <iostream>
using namespace std;

int main()
{
   cout << "ようこそC++へ！¥n";
   cout << "C++をはじめましょう！¥n";

   return 0;
}
```

;（セミコロン）や { }（中カッコ）の位置は正しく入力されているでしょうか？入力が終わったら、第1章で説明した手順にしたがって、コンパイル（・リンク）し、実行してください。画面には、次のように2行表示されるはずです。

Sample1 の実行画面

```
ようこそC++へ！
C++をはじめましょう！
```

画面に出力する

Sample1.cppは、第1章と同じように、画面に文字を表示するC++コードです。画面に文字などを表示することを、プログラミングの世界では、

画面に出力する

と呼んでいます。

そこで本書では、まず最初に、文字列や数値などを画面に「出力する」ためのコードをおぼえることにしましょう。次のコードをみてください。

構文　画面への出力

```
#include <iostream>
using namespace std;

int main()
{
   cout << 出力したい文字列・数値など;

   return 0;
}
```

<<に続けて出力したい文字列・数値などを入力します

このコードが、画面に文字列や数値などを出力するコードの基本形です。引き出し線をつけた行が、コンピュータに対して、画面に出力する処理を指示している部分にあたります。つまり、下線がついた部分に文字列や数値を記述すれば、それが画面に出力されるのです。

「ややこしいコードだなあ・・・」と思う人もいるかもしれません。しかし、いまの段階では、まずC++のコードに慣れることがたいせつです。ここでは、

画面出力のコードはこのように記述するもの

とおぼえてしまってください。このコードは、これから作成するサンプルコードで使っていきます。文字列や数値などのくわしい書きかたについては、2.3節で学ぶことにします。ここでは、おおまかなイメージだけをつかんでおきましょう。

いろいろな出力方法を知る

ちょっとわき道にそれてしまうのですが、画面に文字列を出力するスタイルのコードにもう少し慣れておきましょう。

さきほどのコードの中で、出力したい文字列の前に記述した「cout」という言葉に注目してください。この「cout」とは、

標準出力（standard output）

と呼ばれる、コンピュータの装置と結びつけられている言葉です。

なんだか耳慣れない言葉です。しかし、むずかしいことはありません。「標準出力」とは、「いま使っている、コンピュータの画面装置」のことを意味しています。そして「<<」は、その画面に文字列を出力（＝表示）することを意味しています。つまり、cout・・・の行は、コンピュータに対して、

文字列を「画面」に出力する処理をしなさい

と指示しているのです。

<<という記号を使用すると、文字列や数値を続けて画面に出力することもできます。

ためしに、次のコードを入力してみてください。

Sample2.cpp ▶ ¥nを使う

```
#include <iostream>
using namespace std;

int main()
{
   cout << 1 << 2 << 3 << '¥n' << 4 << 5 << '¥n';

   return 0;
}
```

Sample2の実行画面

```
123
45
```

Sample2プログラムを実行してみましょう、上のように2行出力されたでしょうか？

このコードでは数値を、<<記号を使って続けて出力してみました。ただし、'¥n'と記述した部分は、画面では改行されています。この'¥n'という記号は、**改行を行う**という意味をあらわしています。

coutや<<という言葉については、第16章でくわしく学ぶことにします。

重要

改行をするには、¥nを使う。

標準出力と画面

通常、「標準出力」とは、「画面装置」のことを意味します。ただし、OSの機能によって、標準出力を別の装置に変更することもできます。たとえば、プロンプト上で次のようにプログラムを実行すると、プリンタに文字列が出力されます。標準出力をプリンタに切り替えているのです。

```
Sample1 > PRN ↵
```

このような機能を**リダイレクト**といいます。くわしくは、OSの解説書をお読みください。

2.2 コードの内容

コードの流れをおってみる

それでは、この章の最初で入力したSample1.cppの内容をくわしくみていくことにしましょう。Sample1.cppのコードをじっくりながめてみてください。このコードは、どのような処理をコンピュータに指示しているのでしょうか？

Sample1.cppの内容

```
//画面に文字を出力するコード
#include <iostream>
using namespace std;

int main()
{
    cout << "ようこそC++へ！¥n";
    cout << "C++をはじめましょう！¥n";

    return 0;
}
```


main()関数

私たちはまず、このコードの指示が、どこからはじまり、どこで終わっているのかを知っておく必要があります。最初に

```
int main()
```

と書かれている行をみてください。C++のプログラムは、原則としてこのmain()と記述されている部分から処理が行われることになっています。

次に、Sample1の下から2行目にある

```
return 0;
```

をみてください。この部分の処理が行われると、プログラムが終了します。

中カッコ({})でかこまれた部分は**ブロック**(block)と呼ばれています。このブロックは、プログラムの本体となる重要なもので、**main()関数**(main function)という名前がついています。「関数」という言葉の意味については、第7章でくわしく説明しますので、心にだけとめておきましょう。

```
int main()
{
    ...
    return 0;
}
```

main()関数(プログラムの本体)です

重要

main()関数はプログラムの本体となる。

1文ずつ処理する

それでは、main()関数の中をのぞいてみましょう。C++では、1つの小さな処理(「仕事」)の単位を**文**(statement)と呼び、最後に;(セミコロン)という記号をつけます。そして、この「文」が、

原則として先頭から順番に、1文ずつ処理される

ということになっています。

つまり、このプログラムが実行されると、main()関数内の2つの「文」が、次の順序で処理されるのです。

```
cout << "ようこそC++へ！¥n";      ← 最初に実行されます
    ↓
cout << "C++をはじめましょう！¥n";  ← 次に実行されます
```

cout << ・・・という文は、画面に文字列を出力するためのコードでしたね。そこで、この文が実行されると、画面に2行の文字列が出力されるのです。

重要

文の最後にはセミコロンをつける。
文は原則として先頭から順番に処理される。

```
...
int main()
{
   cout << "ようこそC++へ！¥n";
   cout << "C++をはじめましょう！¥n";

   return 0;
}
```

図 2-1 プログラムの処理
プログラムを実行すると、原則として１文ずつ順番に処理されます。

コードを読みやすくする

ところで、Sample1のmain()関数は、数行分にわたって書かれています。これは、C++のコードでは、

文の途中やブロック中で改行してもよい

ということになっているからです。このため、Sample1のコードでは、main()関数を数行にわけて読みやすくしているのです。

また、C++では、意味のつながった言葉の間などでなければ、自由に空白や改行を入れることができます。つまり、

```
int m ain()
```

などといった表記はまちがいですが、

```
int    main (){
            cout<<
```

というように、空白を入れたり改行してもよいことになっています。

Sample1では、ブロック部分がわかりやすくなるように、「{」の部分で改行し、内部の行の先頭を少し下げているのです。なお、出力する文字列の中で改行することはできません。

コード中で字下げを行うことを、**インデント**（indent）と呼びます。インデントをするには、行頭でスペースキーまたは Tab キーを押してください。

私たちは、これからしだいに複雑なコードを記述していくことになります。インデントをうまく使って、読みやすいコードを書くことを心がけましょう。

```
 ...

int main()
{
  cout << ...

}
```

インデント

図2-2　インデント
ブロック内を字下げして、コードを読みやすくします。

重要　コードを読みやすくするために、インデントや改行を使う。

コメントを記述する

main()関数、つまりプログラムの本体について理解できたでしょうか？　次に、プログラムの本体以外のコードをみてみましょう。まず、main()関数の前に入力した、//記号のある行をみてください。

実は、C++のコンパイラは、

//という記号からその行の終わりまでを無視して処理する

ことになっています。そのため、//記号のあとに、プログラムの実行とは直接関係のない自分の言葉を、「メモ」として入力しておくことができます。これを**コメント**（comment）といいます。通常は、そのコードの内容をメモしておくと便利でしょう。

Sample1では、次のように、コードの先頭でコメントを記述しています。

```
// 画面に文字を出力するコード
```

この部分は無視して処理されます

C++だけでなく、多くのプログラミング言語は、人間にとっては決して読みやすい言語ではありません。このようなコメントを書いておくことによって、読みやすいコードを作成することができるのです。

重要

コメントを使って、コードの内容をわかりやすくする。

もうひとつの方法でコメントを記述する

なお、コメントを記述するには、//という記号のほかに、**/* */**という記号を使うスタイルもあります。

```
/* 画面に文字を
   出力するコード */
```


/*　*/記号の場合は、

/*　*/でかこまれた部分がすべてコメントになる

ということになっています。このため、/*　*/記号を使った場合は、複数行にわたってコメントを書くこともできます。

Sample1のように、//を使うスタイルでは、コメント記号から行の終わりまでを無視するだけなので、複数行にわたってコメントを入力することはできません。C++では、どちらのコメント形式を利用してもかまいません。

ファイルを読み込んでおく

それでは最後に、コードの先頭部分をみてください。

```
#include <iostream>
```

この#ではじまる行は、

画面に表示を行う機能について書かれている「iostream」を、
コンパイルする前に読み込んでおく

という処理をあらわしています。iostreamとは、画面に出力する機能などについて定めたファイルのことをあらわしています。このファイルの読み込み作業を、**インクルード**（include）といいます。

画面に表示するプログラムを作成するときには、必ず**iostream**をインクルードしてください。iostreamを読み込んでおかないと、画面に表示を行うための「cout」という機能を正しく使うことができなくなってしまうからです。

iostreamをあらわすファイルは、C++の開発環境に標準で用意されているため、自分で用意する必要はありません。iostreamのように、前もって読み込まれるファイルのことを**ヘッダファイル**（header file）といいます。なお、#がついた行は、

コンパイラに含まれる**プリプロセッサ**（preprocessor）という特別な部分によって、ほかのコードを翻訳する前に読み込まれることになっています。#がついた行は、最後にセミコロンをつけずに1行で書きます。

さらにコードの2行目をみてください。

```
using namespace std;
```

本来、「cout」はstd.coutと記述するのが正式です。しかし、これを何度も書くのはわずらわしい作業です。そこで、コードの先頭で「using namespace std;」という文を書きます。すると、そのあとは単純に「cout」と書いてよいことになっているのです。

重要

ほかのファイルを読み込むことをインクルードという。
画面に表示をするために、iostreamをインクルードする。

iostream

プリプロセッサの処理

coutについて

Sample1.cpp

```
#include <iostream>
using namespace std;

int main()
{
    cout << ・・・

}
```

コンパイラの処理

図2-3 インクルード
ほかのファイルを読み込むことをインクルードといいます。

2.3 文字と数値

リテラルとは

2.2節では、画面に文字列を出力するかんたんなコードを学びました。そこでこの節では、これまでのコードを応用して、C++の文字・数値・文字列の書き方を学ぶことにしましょう。

まず、次のコードを作成してみてください。

Sample3.cpp ▶ いろいろな値を出力する

```
#include <iostream>
using namespace std;

int main()
{
    cout << 'A' << '¥n';
    cout << "ようこそC++へ!¥n";
    cout << 123  << '¥n';

    return 0;
}
```

文字を出力します
文字列を出力します
数値を出力します

Sample3の実行画面

```
A
ようこそC++へ!
123
```

Sample3では、さまざまな文字や数値が出力されています。このコードの中の'A'や"ようこそC++へ!"、123といった特定の文字や数値の表記を、C++では**リテラル**(literal)と呼んでいます。

リテラルとは、

一定の「値」をあらわすために用いられる、C++の単語のようなもの

と考えてみてください。

リテラルは、次のように4つに分類されています。以下では、これらのリテラルをひとつずつみていくことにします。

- 文字リテラル
- 文字列リテラル
- 数値リテラル
- 論理リテラル（第6章で説明します）

いろいろなトークン

「日本語」や「英語」といった人間の言語が、単語の組みあわせからなりたっているように、C++言語も、単語のような言葉の組みあわせからなりたっています。リテラルは、C++で使われているこの単語のうちのひとつです。

「単語」、つまり、「ある特定の意味をもった文字（またはその組みあわせ）」は、C++では**トークン**（token）と呼ばれています。

トークンは、そのはたらきによって、次のような種類に分類することができます。

- リテラル
- キーワード
- 識別子
- 演算子
- 区切り子（カンマなど）

このうち、リテラルはこの章で学びます。識別子とキーワードは第3章、演算子は第4章で学ぶことにします。

文字リテラル

C++では、

- 1つの文字
- 文字の並び（文字列）

の2種類のリテラルを区別して扱っています。まず、1つの文字をあらわす方法をおぼえましょう。

1つの文字は、**文字リテラル**（character literal）と呼ばれています。これは、

```
'H'
'e'
```

のように、' ' でくくってコード中に記述します。Sample3の中では、'A' が「文字」にあたるわけです。なお、日本語文字の扱いは異なる場合がありますので、ここでは英数字だけを文字として考えておいてください。Sample3の実行結果をみればわかるように、画面には ' ' がつかずに出力されることに注意してください。

重要

1つの文字は ' ' でくくって表記する。

'H' ←—— 文字

図2-4 文字
文字をあらわすときには ' ' でくくります。

エスケープシーケンス

ところで、文字の中には、キーボードから入力できない、特殊な文字もあります。このような文字については、¥を最初につけた2つの文字の組みあわせで「1文字分」をあらわすことができます。これを**エスケープシーケンス**（escape sequence）といいます。エスケープシーケンスには表2-1のようなものがあります。

表2-1：エスケープシーケンス

エスケープシーケンス	意味している文字
¥a	警告音
¥b	バックスペース
¥f	改ページ
¥n	改行
¥r	復帰
¥t	水平タブ
¥v	垂直タブ
¥¥	¥
¥'	'
¥"	"
¥?	?
¥ooo	8進数oooの文字コードをもつ文字（oは0～7の数字）
¥xhh	16進数hhの文字コードをもつ文字（hは0～9の数字とA～Fの英字）

なお、お使いの環境によっては、¥が\（バックスラッシュ）として表示される場合もあるので注意してください。

ためしに、エスケープシーケンスを使って、画面に出力するコードを記述してみましょう。次のコードを入力してください。

Sample4.cpp ▶ 特別な文字を出力する

```
#include <iostream>
using namespace std;

int main()
{
    cout << "円記号を表示します。：" << '¥¥' << '¥n';
    cout << "アポストロフィを表示します。：" << '¥'' << '¥n';

    return 0;
}
```

エスケープシーケンスを使います

Sample4の実行画面

```
円記号を表示します。：¥
アポストロフィを表示します。：'
```

「¥¥」や「¥'」と記述した部分が、「¥」や「'」として出力されているのがわかります。

重要

エスケープシーケンスを使うと、特殊な文字をあらわすことができる。

'¥¥'
↓
¥

図2-5 エスケープシーケンス
特殊な文字をあらわすには、エスケープシーケンスを使います。

文字コード

文字について、もう少しくわしく学んでおきましょう。実はコンピュータの内部では、文字も数値として扱われています。このような各文字のかたちに対応する数値を**文字コード**（character code）といいます。文字コードの種類にはいろいろなものがありますが、代表的なものとしてシフトJISコードやUnicodeがあります。どの文字コードが使われるかは、使っている環境によって異なります。

エスケープシーケンスを使って「¥ooo」や「¥xhh」（表2-1）を出力すると、指定した文字コードに対応する文字が出力されます。次のコードを実行してみましょう。

Sample5.cpp ▶ 文字コードを使う

```
#include <iostream>
using namespace std;

int main()
{
   cout << "8進数101の文字コードをもつ文字は" << '¥101' << "です。¥n" ;
   cout << "16進数61の文字コードをもつ文字は" << '¥x61' << "です。¥n" ;

   return 0;
}
```

文字コードを指定します

シフトJISコードで文字が処理されている場合には、次のように出力されます。

Sample5の実行画面（シフトJISコード）

```
8進数101の文字コードをもつ文字はAです。
16進数61の文字コードをもつ文字はaです。
```

シフトJISコードでは8進数の「101」がA、16進数の「61」がaに対応しています。そのため、上のように「A」や「a」が出力されるのです。このプログラムは、お使いの環境が別の文字コードを使用している場合には、上のように出力されない場合があるので注意してください。なお、8進数や16進数については、この章の最後のコラムで説明しています。

重要

文字コードを指定して、文字を出力することができる。

'¥101'
↓
A

図2-6 文字コード
文字をあらわすには、文字コードを指定する方法もあります。

文字列リテラル

文字1文字に対して、複数の文字の並びを**文字列リテラル**（string literal）といいます。C++では、文字列は文字と異なり、' 'ではなく" "でくくって記述します。たとえば、下のような表記が文字列です。

```
"Hello"
"Goodbye"
```

画面に出力された文字列は、" "がつきません。
文字列については、第9章でさらにくわしく学ぶことにします。

"Hello" ←— 文字列

図2-7 文字列
文字列をあらわすときには、" " でくくります。

重要

文字列は" " でくくって表記する。
文字と文字列の扱いは異なる。

数値リテラル

C++のコードには、数値を表記することもできます。数値には、次のような種類があります。

- 整数リテラル（integer literal）　　1、3、100など
- 浮動小数点数リテラル（floating literal）　　2.1、3.14、5.0など

数値のリテラルは' 'や" " ではくくらないで記述することに注意してください。

整数リテラルには、一般的な数値の表記法以外にも、いろいろな書きかたがあります。たとえば、8進数、16進数で数値を表記することもできます。

- 8進数 …… 数値の先頭に0をつける
- 16進数 …… 数値の先頭に0xとつける

つまり、C++では、次のような方法で数値を表記できるのです。

```
10      10進数の「10」です。10をあらわします
010     8進数の「10」です。8をあらわします
0x10    16進数の「10」です。16をあらわします
0xF     16進数の「F」です。15をあらわします
```

では、いろいろな表記方法を使って、数値を出力してみましょう。

Sample6.cpp ▶ 10進数以外で表記する

```
#include <iostream>
using namespace std;

int main()
{
   cout << "10進数の10は" << 10 << "です。¥n";
   cout << "8進数の10は" << 010 << "です。¥n";
   cout << "16進数10は" << 0x10  << "です。¥n";
   cout << "16進数のFは" << 0xF  << "です。¥n";

   return 0;
}
```

10進数以外の表記を使っています

Sample6の実行画面

```
10進数の10は10です。
8進数の10は8です。
16進数の10は16です。
16進数のFは15です。
```

10進数、8進数、16進数で表記した「10」を出力してみました。10進数であらわすと、8進数の「10」は8、16進数の「10」は16です。数値は、" "の中に入れていません。いろいろな数値の表記法を使っても、画面には10進数で出力されていますね。

10 → 10
010 → 8
0x10 → 16

図2-8 10進数以外の表記
8進数や16進数で整数をあらわすこともできます。

2進数、8進数、16進数

私たちは日ごろ、0から9までの数字で数をあらわしていますね。この数の表記方法を「10進数」といいます。一方、コンピュータの内部では0と1とだけを使った、「2進数」という表記方法で数をあらわしています。

10進数では、

0, 1, 2, 3 ･･･

というように、数値をあらわしていきます。しかし、2進数では同じ数値を

0, 1, 10, 11 ･･･

とあらわすことになっています。10進数では、0から9までの数を使うので、9の次でケタが繰りあがりますが、2進数では、0と1しか使わないので、1の次でケタが繰りあがることになります。

10進数 8 9 10 （繰りあがる）　**2進数** 0 1 10 （繰りあがる）

このため、2進数で表記した数値は、大きなケタになりがちです。たとえば、10

進数での20を2進数であらわすと、

10100

という大きなケタになってしまいます。

そこで、C++では2進数との変換に都合のよい8進数、16進数が、10進数とあわせてよく用いられています。8進数は0から7までの数字、16進数は0から9までとAからFまでの文字を使う方法です。

次の表は10進数の数を2進数、8進数、16進数で表記した場合の対照表です。それぞれ、どの部分でケタが繰りあがるのかに注意してください。

10進数	2進数	8進数	16進数
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14

2.4 レッスンのまとめ

この章では、次のようなことを学びました。

- main( )関数は、C++ プログラムの本体となります。
- 文は、処理の小さな単位となります。
- ブロックは、処理の大きな単位となります。
- コメントとして、コード中にメモを書いておくことができます。
- プリプロセッサは、コンパイラの処理の前に準備作業を行います。
- リテラルには、文字・文字列・数値などがあります。
- 文字リテラルは、' ' でくくります。
- 特殊な文字は、エスケープシーケンスであらわします。
- 文字列リテラルは、" " でくくります。
- 整数リテラルは、8 進数や 16 進数であらわすこともできます。

ここまでに学んだことを使うと、一定の文字や数値を画面に表示させるコードを書くことができます。しかし、これだけの知識では、まだまだ変化にとんだプログラムを作成することはできません。次の章では「変数」という機能を使って、より柔軟なプログラムを作成する方法を学びます。

練習

1. 次のコードはどこかまちがっていますか？ 誤りがあれば、訂正してください。

```
#include<iostream>
using namespace std;int main(){cout<<"こんにちは¥n";
cout<<"さようなら¥n";return 0;}
```

2. 次のコードの適切な部分に「123と45をわけて表示する」というコメントを入れてください。

```
#include <iostream>
using namespace std;

int main()
{
    cout << 1 << 2 << 3 << '¥n' << 4 << 5 << '¥n';

    return 0;
}
```

3. 文字や数値などを使って、次のように画面に出力するコードを記述してください。

```
123
¥100もらった
またあした
```

4. 次のように画面に出力するコードを記述してください。8進数・16進数を使って2とおりのコードを記述してください。

```
6
20
13
```

Lesson 3

変数

第2章では、文字や数値を画面に出力する方法を学びました。文字や数値は、プログラムをはじめたばかりの方にとっても、それほど違和感なくなじめるものだったと思います。この章では、いよいよC++のプログラミング言語らしい機能を学んでいきます。まず、最も基本となる「変数」についてみていくことにしましょう。

Check Point!

- 変数
- 識別子
- 型
- 宣言
- 代入
- 初期化
- const
- 定数

3.1 変数

変数のしくみを知る

プログラムを実行するとき、プログラムはコンピュータにいろいろな値を記憶させながら処理をしていきます。たとえば、

ユーザーが入力した数値を画面に出力する

というかんたんなプログラムについて考えてみてください。私たち人間であれば、本屋などのお店で商品の値段（＝数値）をおぼえ、あとからその値段を紙に書き出すといったことができます。

これと同じように、コンピュータも、数値をどこかに「記憶」しておき、画面に出力することができます。このように、値を記憶する機能は**変数**（variable）と呼ばれています。

コンピュータは、いろいろな値を記憶しておくために、内部に**メモリ**（memory）という装置をもっています。「変数」とは、メモリを利用して値を記憶するしくみなのです。

変数のイメージをつかむために、下の図をみてください。変数は下の図のハコのようなものと考えることができます。変数を使うと、あたかも

変数というハコの中に値を入れる

ように、特定の値をメモリに記憶させることができます。

それでは、変数の機能をくわしくみていくことにしましょう。

図3-1 変数
変数にはいろいろな値を記憶させることができます。

3.2 識別子

変数の「名前」となる識別子

コードの中で変数を扱うには、まず最初に次の2つのことを決めておかなければなりません。

1. 変数に「名前」をつける
2. 変数の「型」を指定する

最初に、1について説明しましょう。

変数を使うには、変数のためにある特定の「名前」を考えておく必要があります。変数の名前は、私たちが適当に選んで決めることができます。

たとえば、「num」という文字を思いついたとします。このような文字の組みあわせは、変数の名前とすることができます。変数の名前として使える文字や数字の組みあわせ（トークン、第2章参照）を、**識別子**（identifier）と呼んでいます。numは識別子のひとつです。

ただし、識別子には次のような規則があります。

- 英字・数字・アンダースコア（_）のいずれかを用います。特殊な記号を含めることはできません。
- 長さには制限がありません。ただし、環境によっては31文字までとなっているものもあります。
- あらかじめC++が予約している「キーワード」を使用することはできません。主なキーワードとして、returnがあります。
- 数字ではじめることはできません。
- 大文字と小文字は異なるものとして区別されます。

num

図3-2 変数に名前をつける
変数の名前には識別子を使います。

上の規則にあてはまる識別子の例をいくつかあげておきましょう。次のような名前を変数の名前とすることができます。

```
a
abc
ab_c
F1
```

一方、次のものは識別子として正しくありません。これらは変数の名前として使うことはできません。どこが誤っているのかを確認してみてください。

```
12a
return
is-a
```


変数の名前は、識別子の規則にあっていれば自由につけることができます。ただし、どのような値を記憶する変数なのか、はっきりわかる名前をつけるようにしましょう。

重要

変数名には識別子を使う。わかりやすい名前をつけること。

3.3 型

型のしくみを知る

次に、変数の「型」について学んでいきましょう。変数には値を記憶させることができます。この値には、いくつかの「種類」があります。値の種類は、**データ型**（data type）、または**型**と呼ばれています。次の表をみてください。C++には、このような基本的な型（基本型）があります。

表3-1：C++の主な基本型の例

種類	名前	サイズ例	記憶できる値の範囲例
論理型	bool	1バイト	trueまたはfalse
文字型	char	1バイト	英数字1文字 −128～127
	unsigned char	1バイト	英数字1文字（符号なし）0～255
整数型	short int	2バイト	整数 −32768～32767
	unsigned short int	2バイト	整数（符号なし） 0～65535
	int	4バイト	整数 −2147483648～2147483647
	unsigned int	4バイト	整数（符号なし） 0～4294967295
	long int	4バイト	長整数 −2147483648～2147483647
	unsigned long int	4バイト	長整数（符号なし） 0～4294967295
浮動小数点型	float	4バイト	単精度浮動小数点数 3.4E−38～3.4E+38
	double	8バイト	倍精度浮動小数点数 1.7E−308～1.7E+308
	long double	8バイト	拡張倍精度浮動小数点数 1.7E−308～1.7E+308

3

-32768～32767まで
の範囲の整数のうちの
どれか1つの値

short int型

図3-3 型
変数を使うには、型を指定する必要があります。

変数を使うためには、どの型の値を記憶させるための変数なのかを、あらかじめ決めておく必要があります。

たとえば、図3-3をみてください。この図は

short int型という種類の値を記憶できる変数

をあらわしています。このような変数を使うと、

-32768～32767までの範囲の整数のうち、どれか1つの値

を記憶させることができるのです。

short int型の変数には、小数点以下のケタをもつ「3.14」などといった値を記憶することはできません。このような値を格納する場合には、小数点以下の数値をあらわせるdouble型などの変数を使わなければなりません。

また、表3-1の「サイズ例」という欄をみてください。これは、「記憶する値がどのくらいのメモリを必要とするか」ということをあらわしています。一般的に、サイズが大きいほど、あらわすことができる値の範囲は広くなります。たとえば、double型の値は、int型の値より多くのメモリを必要としますが、あらわすことができる値の範囲は広くなっています。

なお、C++の基本型のサイズは環境によって異なる場合があります。表3-1に示した「サイズ例」と「記憶できる値の範囲例」は例として考えてください。詳細については、使っているツールのマニュアルを参照するようにしてください。

ビットとバイト

型のサイズと値の範囲には深い関係があります。第２章でも説明したように、コンピュータは０と１を使った２進数の数値で数を認識しているのでしたね。２進数の値の「１ケタ分」は、**ビット** (bit) と呼ばれています。つまり、次のような数値の１ケタが１ビットにあたります。

00101110 ── 1ビット

１ビットでは、２進数の数値の１ケタ分である「0」か「1」のどちらかの値をあらわすことができます。

また、２進数の８ケタの数値は「バイト」(byte) と呼ばれています。つまり、１バイトは８ビットにあたります。１バイトは、2^8=256とおりの値をあらわすことができます。

1ビット　1ビット

00101110 ── 1バイト(8ビット)

さて、表3-1をみてください。「2バイトのshort int型の値」とは、コンピュータの内部では、次のような２進数16ケタの数値となっています。

1バイト　1バイト

0000000000000011 ── 2バイトのshort型の数値例です

これを数えてみると、この２進数16ケタでは、2^{16}=65536とおりの値をあらわすことができることがわかります。

short int型では、この65536種類の値で、私たちが普段使う10進数の-32768から32767までの範囲の値に対応させ、次のようにあらわします。

表：short int 型の値の例

コンピュータの内部（2進数）	あらわしている数値（10進数）	
0000000000000000	0	正の整数に対応しています
0000000000000001	1	
0000000000000010	2	
:	:	
0111111111111111 →	32767	
1000000000000000	-32768	負の整数に対応しています
1000000000000001	-32767	
:		
1111111111111111	-1	

先頭の1ビット分が、数値の正負に対応していることに注意してください。正の数では先頭が0、負の数では1となっています。

同じ2バイトの型でも、対応が異なれば、別の値をあらわすこともできます。たとえば、unsigned short int型という2バイトの型をみてください。これは65536種類の正数（0～65535）をあらわす型です。たとえば、次のように数値をあらわします。

表：unsigned short int型の例

コンピュータの内部では（2進数）	あらわしている数値（10進数）	
0000000000000000	0	正の整数に対応しています
0000000000000001	1	
0000000000000010	2	
:		
1111111111111111 →	65535	

今度は、先頭のビットが0でも1でも、あらわしている数値は正の数になっていますね。

3.4 変数の宣言

変数を宣言する

変数の名前と型が決まったら、さっそくコードの中で変数を使ってみましょう。まずはじめに、「変数を用意する」という作業が必要です。この作業は、

変数を宣言する (declaration)

と呼ばれています。変数は、次のように宣言します。

構文　変数の宣言

```
型名 識別子;
```

ここで、3.2節と3.3節で説明した「型」と「識別子（ここでは変数名）」を指定するわけです。変数の宣言は1つの文で行うので、最後にセミコロン（;）をつけるようにしてください。実際に変数を宣言するコードは、次のようになります。

```
int num;
char c;
double db, dd;
```

① int型の変数numです
② char型の変数cです
③ double型の2つの変数dbとddです

①は、int型の変数numを宣言した文です。②は、char型の変数cを宣言した文です。③は、double型の変数dbとddを2つまとめて宣言したものです。このように、変数は、カンマ（,）で名前を区切り、1つの文の中にまとめて宣言することもできます。

変数を宣言すると、その名前の変数をコードの中で使っていくことができるよ

うになります。

重要

変数は型と名前を指定して宣言する。

図 3-4 **変数の宣言**
変数を宣言すると、変数が用意されます。

変数の宣言をくわしくみると

変数を用意する「宣言」とは、厳密には、次のような２つの処理のことを意味しています。

① 変数の名前と型をコンパイラに知らせる
② 変数のためのメモリを用意する

このうち、②の作業だけをとりあげて、「変数の定義」と呼ぶ場合があります。上で紹介した変数の宣言は、①と②の両方を行うものであるため、このような宣言を「『定義』を含む宣言」と呼ぶこともあります。

3.5 変数の利用

変数に値を代入する

変数を宣言すると、変数に特定の値を記憶させることができます。このことを、

値を代入する（assignment）

といいます。図3-5をみてください。「値の代入」は、用意した変数のハコに、特定の値を入れる（格納する、記憶する）というイメージです。

代入をするには、次のように「=」という記号を使って記述します。

```
num = 3;
```

ちょっとかわった書きかたですね。しかし、これで、変数numに値3を記憶させることができます。この=記号は、値を記憶させる機能をもっているのです。

図3-5 変数への代入
① 変数numを宣言します。
② 変数numに3という値を代入します。

変数に値を代入するコードのスタイルは、次のようになります。

構文 **変数への代入**

```
変数名 = 式;
```

「式」については、次の章でくわしく解説します。ここでは、3や5といった一定の「値」と考えておいてください。

それでは、実際にプログラムを作成して、変数を使ってみることにしましょう。

Sample1.cpp ▶ 変数を使う

```
#include <iostream>
using namespace std;

int main()
{
    int num;                                  ① 変数numを宣言します

    num = 3;                                  ② 変数numに3を代入します

    cout << "変数numの値は" << num << "です。¥n";

    return 0;                                 ③ 変数numを出力します
}
```

Sample1 の実行画面

```
変数numの値は3です。
```

変数numの値が出力されます

このコードでは、まず、①のところでint型の変数numを宣言しています。そして、②のところで変数numに3という値を代入しています。

このように、「=」という記号は、数学の式で使われるような、「●と○が等しい」という意味ではありません。「値を代入する」という機能をあらわすものなのです。注意しておきましょう。

重要

変数には、=を使って値を代入する。

変数の値を出力する

最後に、③の部分をみてください。ここで、変数numの値を出力しています。変数の値を出力するためには、' ' や " " といった引用符をつけないで変数名を記述してください。そうすると、プログラムを実行したときに実際に出力されるのは、

```
num
```

という変数名でなく、変数numの中に格納されている

```
3
```

という値になります。

これで変数の値を画面に出力するコードが記述できました。

3

3

num

図3-6 変数の出力
変数を出力すると、変数に記憶されている値が表示されます。

変数を初期化する

ところで、Sample1では、

```
int num;     ← 変数を宣言します
num = 3;     ← 次の文で変数に値を代入します
```

というように、変数を宣言してから、もうひとつ文を記述して、変数の中に値を代入しました。しかし、C++では、

変数を宣言したとき、同時に、変数に値を格納する

という書きかたもできます。このような処理を、

変数を初期化する（initialization）

といいます。変数を初期化するコードは、次のように書きます。

```
int num = 3;     ← 変数を3で初期化しています
```

この文は、Sample1で2文にわたってあらわした処理を、1文でまとめて行うものなのです。変数の初期化方法をおぼえておきましょう。

構文　変数の初期化

```
型名 識別子 = 式;
```

実際にコードを書くときには、できるだけこのように、変数を初期化するコードを記述したほうが便利でしょう。変数の宣言・代入を2文にわけておくと、値を代入する文を書き忘れてしまうことがよくあるからです。

重要

変数を初期化すると、宣言と値の格納が同時に行える。

代入と初期化は異なるもの

C++では、代入も初期化のどちらも＝という記号を使います。どちらの方法でも変数に値を格納できるのです。しかし厳密には、C++では代入と初期化は異なるものとして区別されています。本書でも、第14章では代入と初期化を厳密に区別しています。

変数の値を変更する

第2章でみてきたように、コード中では、文が1つずつ処理されるのでした。この性質を使って、いったん代入した変数の値を新しい値に変更することができます。次のコードをみてください。

Sample2.cpp ▶ 変数の値を変更する

```
#include <iostream>
using namespace std;

int main()
{
    int num = 3;

    cout << "変数numの値は" << num  << "です。¥n";

    num = 5;

    cout << "変数numの値を変更しました。¥n";
    cout << "変数numの新しい値は" << num  << "です。¥n";

    return 0;
}
```

① 変数の値を出力します
② 変数に新しい値を代入します
③ 変数の新しい値を出力します

Sample2の実行画面

```
変数numの値は3です。
変数numの値を変更しました。
変数numの新しい値は5です。
```

変数の新しい値が出力されます

Sample2では、はじめに変数numを3で初期化し、①のところで出力しています。そして、次に②のところで変数に新しい値として5を代入しています。このように、変数にもう一度値を代入をすると、

値を上書きし、変数の値を変更する

という処理ができるのです。

②で変数の値が変更されたので、③で変数numを出力するときには、新しい値である「5」が出力されています。①と③は同じコードにもかかわらず、変数の値の違うために、画面に出力される値が異なっていることに注意してください。

このように、変数は、いろいろな値を記憶させることができます。"変"数と呼ばれる理由がわかったでしょうか。

図3-7 変数の値を変更する
変数numにもう一度値を代入すると、変数の値が変更されます。

ほかの変数の値を代入する

値を代入するときに=の右辺に記述できるのは、3や5といった一定の数値だけではありません。次のコードを入力してみてください。

Sample3.cpp ▶ ほかの変数の値を代入する

```
#include <iostream>
using namespace std;

int main()
{
    int num1, num2;

    num1 = 3;

    cout << "変数num1の値は" << num1  << "です。¥n";

    num2 = num1;   ← 変数num1の値を変数num2に代入します

    cout << "変数num1の値をnum2に代入しました。¥n";
    cout << "変数num2の値は" << num2  << "です。¥n";

    return 0;
}
```

Sample3の実行画面

```
変数num1の値は3です。
変数num1の値を変数num2に代入しました。
変数num2の値は3です。   ← 変数num2の値は変数num1の値と同じになりました
```

ここでは、=記号の右側に数値ではなく、変数のnum1を記述しています。すると、変数num2には、「変数num1の値」が代入されることになります。画面をみると、たしかに、変数num2に変数num1の値である3が格納されていることがわかりますね。このように、

変数の値をほかの変数に代入する

ということもできるのです。

図3-8 ほかの変数の値の代入
変数num1の値を変数num2に代入することができます。

値の代入についての注意

変数に値を代入するときには、注意しなければならないことがあります。次のコードをみてください。

Sample4.cpp ▶ 値の代入

```
#include <iostream>
using namespace std;

int main()
{
   int num1;
   double num2;

   num1 = 3.14;   // int型の変数に代入します
   num2 = 3.14;   // double型の変数に代入します

   cout << "変数num1の値は" << num1  << "です。¥n";
   cout << "変数num2の値は" << num2  << "です。¥n";

   return 0;
}
```

Sample4の実行画面

```
変数num1の値は3です。
変数num2の値は3.14です。
```

int型の変数には3.14を格納できません

double型の変数は3.14を格納できます

このコードでは、変数num1とnum2に同じ「3.14」を代入しているにもかかわらず、1行目では異なる値の「3」が出力されてしまっています。これは、int型の変数であるnum1には、整数値しか格納できないことになっているからです。

3.3節でも説明したように、変数は、宣言した型によって、記憶できる値の種類が決まります。整数型の変数に小数値を代入すると、自動的に型の変換が行われて、小数点以下が切り捨てられてしまうことになっているのです。

変数の型には注意するようにしてください。型の変換については第4章でも説明しましょう。

図3-9 値の代入に注意
整数型の変数は、小数値をそのまま記憶することはできません。

変数の宣言位置についての注意

しばらくの間、本書では変数の宣言を

main()関数のブロック内

に記述していくことにします。

```
int main()
{
   ...
}
```

この部分で変数を宣言します

実は、main()関数のブロックの外に変数を宣言することもできるのですが、この方法については、第10章でくわしくみていくことにします。なお、同じブロックの中では名前が重複する変数を宣言することはできないので注意してください。

変数の宣言位置に気をつける

C++では変数を使う前であれば、関数の中のどこにでも、宣言を記述することができます。C++のもとになっているC言語では、ブロックの最初の部分に、変数をまとめて宣言しなければならないことになっていました。この点は、C++のほうが自由に変数を記述できるようになっています。

3.6 キーボードからの入力

キーボードから入力する

この章の応用として、ユーザーにキーボードから数値を入力させて、その値を出力するコードを記述してみましょう。キーボードからの入力を受けつける方法を学ぶと、さらに柔軟なプログラムが記述できるようになります。

キーボードから入力するコードは、次のようなスタイルで記述します。

構文　キーボードからの入力

```
#include <iostream>
using namespace std;

int main()
{
    変数の宣言;
    cin >> 変数;
    ...
}
```

（変数：キーボードからの入力を変数に読み込みます）

キーボードからの入力を受けつけるには、**cin >>**という言葉を使います。このスタイルのプログラムでは、cin>>・・・の部分の処理が行われると、実行画面が**ユーザーからの入力を待つ状態**で止まります。そこで、ユーザーは、数値などをキーボードから入力し、Enter キーを押します。すると、入力した値が変数に読み込まれるのです。ただし、空白などの入力は読み飛ばされます。

では、実際にプログラムを作成してみましょう。

Sample5.cpp ▶ 数値を入力する

```
#include <iostream>
using namespace std;

int main()
{
    int num = 0;

    cout << "整数を入力してください。¥n";

    cin >> num;

    cout << num << "が入力されました。¥n";

    return 0;
}
```

キーボードからの入力を促すメッセージを出力します

キーボードからの入力した数値を変数numに読み込みます

入力した数値を出力します

Sample5の実行画面

```
整数を入力してください。
10↵
10が入力されました。
```

キーボードから入力すると・・・

入力した数値が出力されます

このプログラムを実行すると、「整数を入力してください」というメッセージが画面に出力されます。そして、コンピュータはユーザーからの入力を待つ状態になります。

ここで「10」と入力し、Enter キーを押してみましょう。すると、画面に「10が入力されました。」と出力されるはずです。

何度もプログラムを実行し、いろいろな数値を入力してみてください。このコードを使うと、さまざまな数値を出力することができるはずです。

図3-10 キーボードからの入力を読み込む手順
キーボードから入力を受けつけて、変数に読み込みます。

2つ以上の数値を入力する

>>記号を続けて記述すると、2つ以上の数を続けて入力することができます。今度は、次のコードを記述してみましょう。

Sample6.cpp ▶ 2つ以上の数値を続けて入力する

```
#include <iostream>
using namespace std;

int main()
{
    int num1, num2;

    cout << "整数を2つ入力してください。¥n";

    cin >> num1 >> num2;

    cout << "最初に" << num1 << "が入力されました。¥n";
    cout << "次に" << num2 << "が入力されました。¥n";

    return 0;
}
```

2回続けて入力させます

Sample6の実行画面

```
整数を2つ入力してください。
5 ↵
10 ↵
最初に5が入力されました。
次に10が入力されました。
```

2回続けて入力します

このプログラムを実行すると、キーボードから「5」や「10」という2つの数を続けて入力することができます。

最初に入力した「5」は変数num1、あとから入力した「10」は変数num2に読み込まれます。最後にnum1とnum2を出力するコードを記述しているので、入力した2つの数が出力されています。

標準入力のしくみを知る

ところで、coutが「標準出力」をあらわしていたように、cinは**標準入力**（standard input）という概念をあらわしています。通常、「標準入力」とはコンピュータの「キーボード」のことをさしています。

ここで使っている>>記号は、**キーボードからの入力を指定した変数に送りこむ**という機能をもっています。キーボードから入力を行う場合にも、画面に出力するときと同じように、コードの最初にiostreamを読み込むようにしてください（第2章）。

誤った入力

ところで、もしユーザーが誤った値を入力した場合にはどうなるのでしょうか？　たとえば、整数を入力するプログラムに対して、ユーザーが小数を入力してしまった場合はどうなるのでしょうか？

ユーザーが誤った入力をした場合には、正しい表示が行われなかったり、予想外のエラーがおきたりします。

実際のプログラムを作成する際には、ユーザーが誤った入力をした場合の処理も記述しておかなければなりません。本書では、いろいろな状況に応じて処理を行う方法を第５章で学びます。実際のプログラムを作成するにあたっては、ユーザーの入力ミスを考えて、コードを記述しなければならないことを忘れないでください。

3.7 定数

const を指定する

3.5節では変数の値を変更する方法を学びました。しかし、変数を初期化するときに特別な指定をすると、その変数の値を変更できなくすることができます。次のコードをみてください。

Sample7.cpp ▶ 定数を利用する

```
#include <iostream>
using namespace std;

int main()
{
    const double pi = 3.1415;

    cout << "円周率の値は" << pi << "です。¥n";
    cout << "円周率の値は変更しません。¥n";

    //このような代入による変更はできません。
    //pi = 1.44;

    return 0;
}
```

① const を指定して pi を初期化します

② pi の値を変更できなくなります

Sample7 の実行画面

```
円周率の値は3.1415です。
円周率の値は変更しません。
```

このコードの中の②の部分のコメントをはずすと、このプログラムはコンパイ

ルすることができなくなります。これは、①の部分で、constという指定をしているからです。

```
const double pi = 3.1415;
```

constを指定して初期化します

このように、constを指定して変数を初期化すると、これよりあとのコード中では、変数piに値を代入できなくなるのです。

constのしくみを知る

どうしてこのような指定をする必要があるのでしょうか？

円周率は、「3.1415」という数値としてコードの中に記述することができます。しかし、円周率ならば「pi」という言葉で記述したほうがわかりやすいでしょう。

そこで、コード中では「pi」という変数に、3.1415を代入して使いたい気がします。

しかし、変数の値は変更することができます。どこかで誤ってpiの値に「1.44」という値を代入してしまう可能性があります。ところが、constを指定した変数には値を代入できないので、そのような誤りはおきないことになります。

つまり、constを指定すると、

「pi」という言葉を使って、「3.1415」という「一定の」数値をあらわすことができる

ようになるのです。

const指定をした変数は値を変更することができないので、**定数**（constant）とも呼ばれています。定数は次のような初期化によって記述するのです。

構文　**constキーワード**

```
const 型名 識別子 = 式;
```

変数の初期化の文にconstがついているだけですね。

また、プログラムを作成したあとで事情がかわり、もっと精密に計算しなければならなくなったとしましょう。つまり、円周率の値として「3.141592」という数値を使うことになったとしましょう。

このときは、次のように最初の初期化文だけを修正し、もう一度コンパイルします。そうすると、コード中で使っているすべてのpiが新しい値を意味することになります。

```
const double pi = 3.141592;
```

このように変更すれば・・・

コード中で使ったpiはすべて3.141592を意味することになります

定数を使うと、プログラムを変更しやすくなるので、たいへん便利なのです。

constキーワードの注意

すでに説明したように、constキーワードを指定した定数piには、あとから値を代入することができません。次のように値を代入しようとしても、コンパイルするときにエラーが表示されますので、注意してください。

```
const double pi = 3.1415;
pi = 1.44;
```

定数には・・・

値を代入することはできません

また、constを指定した場合は、必ず初期化をする必要があります。初期化をせずに、あとで値を代入しようとしても、やはりコンパイルすることができません。

```
const double pi;
pi = 3.1415;
```

定数を初期化しないで・・・

あとから値を代入することはできません

1.44

×

3.1415

pi

図3-11 const
const指定して初期化した変数には、あとから値を代入することはできません。

3.8 レッスンのまとめ

この章では、次のようなことを学びました。

- 変数には、値を格納することができます。
- 変数は、名前と型を指定して宣言します。
- 変数の「名前」には、識別子を使います。
- 変数に値を代入するには、= 記号を使います。
- 変数を初期化すると、宣言と同時に値を格納できます。
- 変数に値を代入すると、格納されている値が変更されます。
- キーボードから入力をすることができます。
- const を指定した変数は、値を変更できなくなります。

変数はC++の最も基本的な機能です。といっても、この章に登場したサンプルだけでは、変数のありがたみを感じることはむずかしいかもしれません。しかし、たくさんのコードを入力し、本書を読み終えるころには、変数がC++にはなくてはならない機能だということがわかるはずです。さまざまな変数に慣れたあと、この章に戻ってもう一度復習してみてください。

練習

1. 次のように画面に出力するコードを記述してください。

```
円周率の値はいくつですか？
3.14 ↵
円周率の値は3.14です。
```

2. 次のように画面に出力するコードを記述してください。

```
アルファベットの最初の文字は何ですか？
a ↵
アルファベットの最初の文字はaです。
```

3. 次のように画面に出力するコードを記述してください。

```
身長と体重を入力してください。
165.2 ↵
52.5 ↵
身長は165.2センチです。
体重は52.5キロです。
```

Lesson 4

式と演算子

コンピュータはさまざまな処理を行うことができます。このとき必要になるのが「演算」です。C++ のプログラムを作成するときも、演算に関する機能は欠かせません。C++では、演算をシンプルに行うために「演算子」という機能を用意しています。この章では、いろいろな演算子の使いかたについて学びましょう。

Check Point!

- 式
- 演算子
- オペランド
- インクリメント演算子
- デクリメント演算子
- 代入演算子
- 演算子の優先順位
- 型変換
- キャスト演算子

4.1 式と演算子

式のしくみを知る

コンピュータは、いろいろな処理を「計算」することによって行います。そこで、この章では最初に、**式** (expression) というものについて学ぶことにしましょう。「式」を理解するためには、

1+2

といった「数式」を思いうかべればわかりやすいかもしれません。C++では、このような式をコードの中で使っていきます。

C++の「式」の多くは、

演算子 (演算するもの： operator)
オペランド (演算の対象： operand)

を組みあわせてつくられています。たとえば、「1+2」の場合は、「+」が演算子、「1」と「2」がオペランドにあたります。

さらに、式の「評価」というのも重要な概念です。「評価」を知るためには、式の「計算」をイメージしてみてください。この計算が式の評価にあたります。たとえば、1+2が評価されると3になります。評価されたあとの3を「式の値」と呼びます。

図4-1 式
1+2という式は評価されて3という値をもちます。

式の値を出力する

いままで学んできた画面に出力するコードを利用すると、式の値を出力することができます。次のようなコードを入力してみましょう。

Sample1.cpp ▶ 式の値を出力する

```
#include <iostream>
using namespace std;

int main()
{
    cout << "1+2は" << 1+2 << "です。¥n";
    cout << "3*4は" << 3*4 << "です。¥n";

    return 0;
}
```

1+2という式を記述します

Sample1の実行画面

```
1+2は3です。
3*4は12です。
```

式が評価されて3が出力されます

ここでは、「1+2」という式を記述しました。実行画面をみると、「3」が出力されていますね。

次の文も同じように、「3*4」という式を記述しています。これは「3×4」という計算を意味します。C++ではかけ算をするために、*という記号を使うことになっているのです。

このように、画面に出力される値は、式が評価されたあとの値となっていることがわかります。

いろいろな演算をする

式の中で、演算の対象（オペランド）となるものは、1や2といった一定の数ばかりではありません。次のコードを入力してためしてみましょう。

Sample2.cpp ▶ 変数の値を使う

```
#include <iostream>
using namespace std;

int main()
{
   int num1 = 2;
   int num2 = 3;
   int sum = num1+num2;

   cout << "変数num1の値は" << num1 << "です。¥n";
   cout << "変数num2の値は" << num2 << "です。¥n";
   cout << "num1+num2の値は" << sum << "です。¥n";

   num1 = num1+1;

   cout << "変数num1の値に1を足すと" << num1 << "です。¥n";

   return 0;
}
```

①num1 + num2の値をsumに代入します

②num1+1の値をnum1に代入します

Sample2の実行画面

```
変数num1の値は2です。
変数num2の値は3です。
num1+num2の値は5です。
変数num1の値に1を足すと3です。
```

たし算の結果が出力されます

このコードでは、①と②の部分で、変数をオペランドとして使った式を記述しています。このように、一定の値だけでなく、変数もオペランドとすることができるのです。ひとつずつみていきましょう。

まず、①のsum = num1 + num2は、

変数num1と変数num2に記憶されている「値」どうしのたし算を行い、その値を変数sumに代入する

という演算を行う式です。

次に、②のnum1 = num1 + 1は、

変数num1の値に1をたし、その値を再度num1に代入する

という式です。右辺と左辺がつりあっていない、かわった表記にもみえます。けれども、=の記号は「等しい」という意味ではなく、「値を代入する」という機能をもつものでしたね。そこで、このような記述が可能なのです。

図4-2 **sum=num1+num2（左）num1=num+1（右）**
変数に記憶されている値をたし算することもできます。

キーボードから入力した値をたし算する

さて、「変数を使った式」について、ちょっと考えてみてください。第3章でみたように、変数にはいろいろな値を格納できます。つまり、変数を使った場合の式の値は、

コードが処理されるときの変数の値によって異なってくる

ということがわかります。このことを利用すれば、さらにバリエーションにとんだ

プログラムをつくることができるようになります。次のようなコードを入力してみましょう。

Sample3.cpp ▶ たし算プログラム

```
#include <iostream>
using namespace std;

int main()
{
   int num1, num2;

   cout << "整数1を入力してください。¥n";
   cin >> num1;

   cout << "整数2を入力してください。¥n";
   cin >> num2;

   cout << "たし算の結果は" << num1+num2 << "です。¥n";

   return 0;
}
```

入力した数を変数num1、num2に記憶させます

num1とnum2の値をたし算した結果を出力します

Sample3の実行画面

```
整数1を入力してください。
5 ↵
整数2を入力してください。
10 ↵
たし算の結果は15です。
```

入力した数をたし算した結果が出力されます

Sample3は、キーボードから入力した値を変数に読み込み、たし算するコードです。第3章で学んだ、キーボードからの入力を受けつけるコードを使っています。プログラムを実行し、いろいろな整数を入力すると、入力された数がたし算されて出力されるはずです。

このように、変数と演算子を使ってコードを記述すれば、プログラムを実行したときの状況に応じたプログラムを作成できます。これまでは、いつも同じ文字や数値しか出力することができませんでしたが、今度は入力した値によって、違

う計算結果を出力することができるようになりました。バリエーションにとんだプログラムを作成することができたわけです。

図 4-3 **キーボードからの入力をたし算する**
いろいろな数値を入力して、たし算をすることができます。

いろいろな式

式は、

```
1+2
3*4
```

といった数式のようなものばかりでなく、

num1 ・・・ 変数
5 ・・・ 定数

なども、それだけで「式」であると考えています。

つまり、「5」という式の値は5です。また、「num1」という式の値は、変数num1に5が格納されているときは5、10が格納されているときは10となります。

このような小さな式は、ほかの式と組みあわせて、大きな式をつくります。たとえば、「num1+5」という式の値は、式num1の値と式5の値を+で演算した結果となるわけです。

4.2 演算子の種類

いろいろな演算子

C++には、+演算子以外にもたくさんの演算子があります。演算子の種類を次の表に示しておきましょう。

表4-1：演算子の種類

記号	名前	オペランドの数	記号	名前	オペランドの数
+	加算	2	<=	以下	2
-	減算	2	==	等価	2
*	乗算	2	!=	非等価	2
/	除算	2	!	論理否定	1
%	剰余	2	&&	論理積	2
+	単項＋	1	||	論理和	2
−	単項—	1	*	間接参照	1
~	補数	1	,	順次	2
&	ビット論理積	2	( )	関数呼び出し	2
|	ビット論理和	2	[ ]	配列添字	2
^	ビット排他的論理和	2	::	スコープ解決	2
<<	左シフト	2	.	メンバ参照（ドット）	2
>>	右シフト	2	−>	メンバ間接参照（アロー）	2
++	インクリメント	1	? :	条件	3
--	デクリメント	1	new	動的メモリ確保	1
>	より大きい	2	delete	動的メモリ解放	1
>=	以上	2	sizeof	サイズ	1
<	未満	2			

演算子には、オペランドを1つとるもの、2つとるものなどがあります。オペランドが1つの演算子のことを**単項演算子**（unary operator）と呼びます。また、オペ

ランドが2つの演算子は**2項演算子**、オペランドが3つの演算子は**3項演算子**と呼びます。

たとえば、次のようにひき算を行う-演算子は、オペランドを2つとる演算子です。

```
10-2
```

一方、「負の数」をあらわすために使う-演算子は、オペランドを1つとる演算子です。

```
-10
```

それでは、この表に掲載されているいろいろな演算子を使ったコードを記述してみましょう。

Sample4.cpp ▶ いろいろな演算子を利用する

```
#include <iostream>
using namespace std;

int main()
{
   int num1 = 10;
   int num2 = 5;

   cout << "num1とnum2にいろいろな演算を行います。¥n";
   cout << "num1+num2は" << num1+num2 << "です。¥n";
   cout << "num1-num2は" << num1-num2 << "です。¥n";
   cout << "num1*num2は" << num1*num2 << "です。¥n";
   cout << "num1/num2は" << num1/num2 << "です。¥n";
   cout << "num1%num2は" << num1%num2 << "です。¥n";

   return 0;
}
```

いろいろな演算を行います

Sample4の実行画面

```
num1とnum2にいろいろな演算を行います。
num1+num2は15です。
num1-num2は5です。
num1*num2は50です。
num1/num2は2です。
num1%num2は0です。
```

Sample4では、たし算・ひき算・かけ算・わり算を行っています。それほどむずかしくはありませんね。ただ、最後の「%演算子」(剰余演算子)という演算子には、あまりなじみがないかもしれません。この演算子は、

num1 ÷ num2= ●...あまり×

という計算における「×」を式の値とする演算子です。つまり、%演算子は「あまりの数を求める」演算子というわけです。このコードでは「10 ÷ 5=2 あまり0」ですから、0が出力されています。

num1やnum2の値を変更して、いろいろな演算を行ってみてください。ただし、/演算子では、0によるわり算をすることはできません。

オペランドの型

演算子がどのような演算を行うかということは、演算対象(オペランド)の「型」と密接な関係があります。オペランドの型によっては、使うことのできない演算子もあるので注意してください。このことについては、4.6節で学ぶことにしましょう。

インクリメント・デクリメント演算子

それでは、表4-1の演算子のうち、プログラムを作成するときによく使うものをみていきましょう。まず、表の中にある「++」という演算子をみてください。この演算子は次のように使います。

```
a++;
```
整数aの値を1増やします

++演算子は、**インクリメント演算子**（increment operator）と呼ばれています。「インクリメント」とは、（変数の）値を1増やす演算のことです。つまり、次のコードでは、変数aの値を1増やしていますから、さきほどのコードと同じ処理を行っていることになります。


```
a = a+1;
```
値を1増やす演算は、このようにも書けます

一方、-を2つ続けた「--」は**デクリメント演算子**（decrement operator）といいます。デクリメントは、変数の値を1減らす演算子です。

```
b--;
```
変数bの値を1減らします

このデクリメント演算子は、次のコードと同じ意味になります。

```
a = a-1;
```
値を1減らす演算は、このようにも書けます

図4-4 インクリメントとデクリメント
変数の値に1を加算（減算）するには、インクリメント（デクリメント）演算子を使います。

重要

インクリメント（デクリメント）演算子は、変数の値を1加算（減算）する。

インクリメント・デクリメントの前置と後置

インクリメント・デクリメント演算子は、オペランドの前と後ろのどちらにでも記述することができます。つまり、変数aをインクリメントする場合は、

```
a++
++a
```

という2つの書きかたができるのです。上のようにオペランドの後ろに置く場合を「後置インクリメント演算子」、前に置く場合を「前置インクリメント演算子」と呼びます。変数を1増やすという目的だけなら、どちらの書きかたでも同じことになります。

しかし、この表記の違いによって、プログラムの実行結果が異なる場合もあります。次のコードを記述してみてください。

Sample5.cpp ▶ 前置・後置インクリメント演算子を使う

```
#include <iostream>
using namespace std;

int main()
{
   int a = 0;
   int b = 0;

   b = a++;

   cout << "代入後にインクリメントしたのでbの値は" << b << "です。¥n";

   return 0;
}
```

後置インクリメント演算子を使います

Sample5の実行画面

```
代入後にインクリメントしたのでbの値は0です。
```

ここでは、後置インクリメント演算子を使ってみました。ところが、これを前置インクリメント演算子にすると、異なる実行結果となります。コードの中のインクリメント演算子の部分を次のように前置のものに変更し、もう一度プログラムを作成してみてください。


```
...
b = ++a;
cout << "代入前にインクリメントしたのでbの値は" << b << "です。¥n";
...
```

前置インクリメント演算子を使います

プログラムを実行すると、今度の画面の出力は、

変更後のSample5の実行画面

```
代入前にインクリメントしたのでbの値は1です。
```

となります。

はじめに使った後置インクリメント演算子では、

変数bにaの値を代入してから → aの値を1増やす

という処理を行うのですが、前置インクリメント演算子は、

aの値を1つ増やしてから → 変数bにaの値を代入する

という逆の処理が行われます。このため、出力される変数bの値が異なるのです。ここではコードを省略しますが、デクリメント演算子も同じ性質をもっています。Sample5をみながら、前置・後置デクリメント演算子を使ったコードを作成し、ためしてみてください。

図4-5 インクリメントの前置と後置
後置にすると、代入してから変数の値が増やされます（左）。前置にすると、代入の前に変数の値が増やされます（右）。

インクリメントとデクリメントの使いかた

インクリメント・デクリメント演算子は１つずつ値を増やしたり減らしたりするため、何らかの処理の回数を１回ずつカウントするときなどによく用いられます。第６章で紹介するfor文では、この演算子がよく使われます。

代入演算子

次に、**代入演算子**（assignment operator）について学びましょう。代入演算子は、これまでにも、変数に値を代入する際に使ってきた「=」という記号のことです。この演算子は、通常の=の意味である「等しい」（イコール）という意味ではないことは、すでに説明しました。代入演算子は、

左辺の変数に右辺の値を代入する

という機能をもつ演算子なのです。代入演算子は = だけではなく、=とほかの演算を組みあわせたバリエーションもあります。次の表をみてください。

表4-2：代入演算子のバリエーション

<table>
<tr><th>記号</th><th>名前</th><th>記号</th><th>名前</th></tr>
<tr><td>+=</td><td>加算代入</td><td>&=</td><td>論理積代入</td></tr>
<tr><td>−=</td><td>減算代入</td><td>^=</td><td>排他的論理和代入</td></tr>
<tr><td>*=</td><td>乗算代入</td><td>|=</td><td>論理和代入</td></tr>
<tr><td>/=</td><td>除算代入</td><td><<=</td><td>左シフト代入</td></tr>
<tr><td>%=</td><td>剰余代入</td><td>>>=</td><td>右シフト代入</td></tr>
</table>

これらの代入演算子は、ほかの演算と代入を同時に行うための複合的な演算子となっています。たとえば、+=演算子をみてみましょう。

```
a += b;
```

a+bの値をaに代入します

+=演算子は、

変数aの値に変数bの値をたし算し、その値を変数aに代入する

という演算を行います。+演算子と=演算子の機能をあわせたような機能をもっているのです。

このように、四則演算などの演算子（●としておきます）と組みあわせた複合的な代入演算子を使った文である

```
a ●= b;
```

は、通常の代入演算子=を使って、

```
a = a ● b;
```

と書きあらわすことができます。

つまり、次の2つの文はどちらも、「変数aの値とbの値をたして変数aに代入する」という処理をあらわしています。

```
a += b;
a = a+b;
```

どちらもa+bの値をaに代入する文です

なお、複合的な演算子では、

```
a + = b;
```

ここにスペースをあけることはできません

などと、+と=の間にスペースをあけて記述してはいけません。

図 4-6 複合的な代入演算子
複合的な代入演算子を使うと、四則演算と代入をシンプルに記述することができます。

ためしに、+=演算子を使ってコードを記述してみましょう。

Sample6.cpp ▶ 複合的な代入演算子を使う

```
#include <iostream>
using namespace std;

int main()
{
   int sum = 0;
   int num = 0;
```

```
    cout << "1番目の整数を入力してください。¥n";
    cin >> num;
    sum += num;
    cout << "2番目の整数を入力してください。¥n";
    cin >> num;
    sum += num;
    cout << "3番目の整数を入力してください。¥n";
    cin >> num;
    sum += num;

    cout << "3つの数の合計は" << sum << "です。¥n";

    return 0;
}
```

複合的な代入演算子を使っています

Sample6の実行画面

```
1番目の整数を入力してください。
1 ↵
2番目の整数を入力してください。
3 ↵
3番目の整数を入力してください。
4 ↵
3つの数の合計は8です。
```

このサンプルでは、入力した数値をたし算した結果を、+=演算子を使って順番に変数sumに格納しています。+=演算子を使うことによって、シンプルにコードを書きあらわすことができていますね。同じコードを、+演算子と=演算子を使って書きかえる方法も考えてみてください。

sizeof演算子

今度は、**sizeof演算子**（sizeof operator）という演算子を使ってみましょう。次のようにコードを入力してみてください。

Sample7.cpp ▶ sizeof 演算子を使う

```
#include <iostream>
using namespace std;

int main()
{
    int a = 1;
    int b = 0;

    cout << "short int型のサイズは" << sizeof(short int) <<
        "バイトです。¥n";
    cout << "int型のサイズは" << sizeof(int) << "バイトです。¥n";
    cout << "long int型のサイズは" << sizeof(long int) <<
        "バイトです。¥n";
    cout << "float型のサイズは" << sizeof(float) << "バイトです。¥n";
    cout << "double型のサイズは" << sizeof(double) <<
        "バイトです。¥n";
    cout << "long double型のサイズは" << sizeof(long double)
        << "バイトです。¥n";
    cout << "変数aのサイズは" << sizeof(a) << "バイトです。¥n";
    cout << "式a+bのサイズは" << sizeof(a+b) << "バイトです。¥n";

    return 0;
}
```

型のサイズを調べます

式のサイズを調べます

Sample7 の実行画面

```
short int型のサイズは2バイトです。
int型のサイズは4バイトです。
long int型のサイズは4バイトです。
float型のサイズは4バイトです。
double型のサイズは8バイトです。
long double型のサイズは8バイトです。
変数aのサイズは4バイトです。
式a+bのサイズは4バイトです。
```

sizeof演算子を使うと、いろいろな型や式の値のサイズを知ることができます。実行結果をみると、sizeof(●)と記述した部分では、●の部分の型や式の値のサイズ（表3-1）がバイト単位で出力されているのがわかります。なお、C++の型や式のサイズは実行環境によって異なる場合がありますので、表示されているサイズ

がこの結果と異なる場合もあります。

C++のコードを記述する際には、型や式のサイズを知らなければ処理できない問題にぶつかる場合があります。そのようなときには、sizeof演算子を使って、プログラムを実行している環境での型や式の値のサイズを調べればいいのです。

図4-7 sizeof演算子
型や式のサイズを調べるには、sizeof演算子を使います。

sizeof演算子で、型や式のサイズを調べることができる。

シフト演算子

最後に、もう少し複雑な演算子をとりあげておきましょう。それは、表4-1の<<および>>という記号で示した**シフト演算子**（shift operator）です。

「シフト演算」とは、

数値を2進数であらわした場合のケタを、
左または右に指定数だけずらす（シフトする）

という演算のことです。

たとえば、<<演算子は**左シフト演算子**と呼ばれ、

左辺を2進数で表記したときの値を、右辺で指定したケタ数分だけ左にずらし、
はみだしたケタ数だけ右端から0を入れる

という演算を行います。ちょっと長い言葉になってしまいましたが、実際に例を

みてみましょう。

ここでは、short int型（第3章）の値に対して5 << 2という左シフト演算を行う場合を考えてください。

5<< 2の演算は次のようになります。

```
  5     0000000000000101
<< 2
---------------------------
  20    0000000000010100
```

2ケタ左にずらし、右端のケタから0を入れます

2進数の10100は10進数の20です。つまり、5 << 2は、20となります。

一方、>>演算子は、**右シフト演算**と呼ばれる演算を行います。こちらは、

右辺で指定したケタ数を右にずらし、
はみだしたケタ数だけ左端から0を入れる

という演算です。5>>2の例では次のようになります。

```
  5     0000000000000101
>> 2
---------------------------
  1     0000000000000001
```

2ケタ右にずらし、左端のケタから0を入れます

ただし、右シフト演算の場合は、5のようなオペランドが負の数であった場合は、左端のケタに1を入れる場合もあります。くわしくは、お使いの開発環境の説明書を参考にしてください。

ところで、<<と>>という記号は、これまでにお目にかかっていますね。

サンプルコードの中で使っている

```
cout << ・・・;
cin >> ・・・;
```

のことです。入出力に使われている<<演算子・>>演算子は、ここで説明したシフト演算子とは異なる使いかたをしています。第13章では、演算子に異なる用法を割り当てる方法を学びます。

4.3 演算子の優先順位

演算子の優先順位とは

次の式をみてください。

```
a+2*5
```

2*5 が先に評価されます

この式では、+演算子と*演算子の2つが使われています。1つの式の中で複数の演算子を組みあわせて使うことができます。このとき、式はどのような順番で評価される（演算が行われる）のでしょうか？

通常の四則演算では、たし算より、かけ算のほうを先に計算しますね。これは、数式の規則では、たし算よりもかけ算の演算のほうが

優先順位が高い

からです。C++の演算子の場合も同じです。上のコードでは、「2*5」が評価されてから「a+10」が評価されます。

演算子の優先順位は、変更することもできます。通常の数式と同じようにカッコを使って、カッコ内を優先的に評価させるのです。次の式では、「a+2」が先に評価されてから、その値が5倍されます。

```
(a+2)*5
```

カッコ内が先に評価されます

それでは、ほかの演算子ではどうなるのでしょうか。次の式をみてください。

```
a = b+2;
```

代入演算子のような演算子は、四則演算よりも優先順位が低いため、上の式は次の式と同じ順序で演算されることになります。

```
a = (b+2);
```


C++で使われる演算子の優先順位は、表4-3のようになっています。

図4-8 演算子の優先順位
演算子には優先順位があります。優先順位を変更するには()を使います。

表4-3：演算子の優先順位（実線で区切られた中は同じ優先順位）

記号	名前	結合規則
::	スコープ解決	−
::	グローバル解決	−
()	関数呼び出し	左
[]	配列添字	左
.	メンバ参照	左
->	メンバ間接参照	左
++	後置インクリメント	左
--	後置デクリメント	左

記号	名前	結合規則
!	論理否定	右
~	補数	右
+	単項＋	右
-	単項―	右
sizeof	サイズ	右
++	前置インクリメント	右
--	前置デクリメント	右
&	アドレス	右
*	間接参照	右
new	動的メモリ確保	右
delete	動的メモリ解放	右
()	キャスト	右
%	剰余	左
*	乗算	左
/	除算	左
+	加算	左
-	減算	左
<<	左シフト	左
>>	右シフト	左
>	より大きい	左
>=	以上	左
<	未満	左
<=	以下	左
==	等価	左
!=	非等価	左
&	ビット論理積	左
|	ビット論理和	左
^	ビット排他的論理和	左
&&	論理積	左
||	論理和	左
? :	条件	右
=	代入	右
,	順次	左

同じ優先順位の演算子を使う

ところで、同じレベルの優先順位の演算子が、同時に使われた場合はどうなるのでしょうか？　四則演算では、優先順位が同じ場合、必ず「左から順に」計算する規則になっていますね。このような演算の順序を**左結合**（left associative）といいます。

C++の+演算子も左結合的な演算子です。つまり、

```
a+b+1
```

と記述したときには、

```
(a+b)+1
```


という順序で評価されるわけです。

逆に、右から評価される演算子もあり、これを**右結合**（right associative）といいます。たとえば、代入演算子は右結合的な演算子です。つまり、

```
a = b = 1
```

と記述したときには、

```
a = (b = 1)
```


という順番で右から評価されるため、まず変数bに1が代入され、続いてaにも1という値が代入されます。一般的に、単項演算子と代入演算子は右結合的な演算子となっています。

4.4 型変換

大きなサイズの型に代入する

実は、これまでみてきた演算子とそのオペランドの型には、密接な関係があります。この節ではまず最初に、代入演算子と型の関係についてみていくことにしましょう。次のように、変数に値を代入するコードをみてください。

Sample8.cpp ▶ 大きなサイズの型に代入する

```
#include <iostream>
using namespace std;

int main()
{
    int inum = 160;
    double dnum;

    cout << "身長は" << inum << "センチです。¥n";
    cout << "double型の変数に代入します。¥n";

    dnum = inum;    // 大きなサイズの型に代入します

    cout << "身長は" << dnum << "センチです。¥n";

    return 0;
}
```

Sample8の実行結果

```
身長は160センチです。
double型の変数に代入します。
身長は160センチです。
```

このコードでは、int型の変数の値をdouble型の変数に代入しています。すると、int型の値はdouble型に変換されて代入されることになります。

第3章で、型のサイズについて学んだことを思い出してください。一般にC++では、

大きなサイズの型の変数に小さなサイズの型の値を代入する

ことができます。このように、型が変換されることを**型変換**といいます。

図4-9 大きなサイズの型への代入
大きなサイズの型の変数に、小さなサイズの型の数値を代入することができます。

小さなサイズの型に代入する

それでは逆に、小さなサイズの型の変数に大きなサイズの型の値を代入する場合はどうなるのでしょうか？ 次のコードをみてください。

Sample9.cpp ▶ 小さなサイズの型に代入する

```
#include <stdio.h>
using namespace std;

int main(void)
{
   double dnum = 160.5;
   int inum;
```

```
    cout << "身長は" << dnum << "センチです。¥n";
    cout << "int型の変数に代入します。¥n";

    inum = dnum;   ●── 小さなサイズの型に代入すると・・・

    cout << "身長は" << inum << "センチです。¥n";

    return 0;
}
```


Sample9の実行画面

```
身長は160.5センチです。
int型の変数に代入します。
身長は160センチです。   ●── 値の一部が失われる場合があります
```

今度はさきほどとは逆に、double型の変数の値をint型の変数に代入しています。double型の値がint型に変換されて代入されるわけです。

ただし、小さなサイズの型に変換した場合には、その型であらわせない部分は、上のように切り捨てられることになります。たとえば160.5という値は、int型の変数にそのまま格納することはできません。小数点以下が切り捨てられて160という整数が格納されることになります。

キャスト演算子を使う

型が変換されるときには、

型を変換することを明示的に書いておく

ということができるようになっています。次の**キャスト演算子**（cast operator）という演算子をみてください。

構文　キャスト演算子

```
(型名) 式
```

キャスト演算子は、

指定した式の型を、() 中で指定した型に変換する

という演算を行うものです。

Sample9をキャスト演算子を使って書きかえてみましょう。Sample9で代入をしている部分を、次のように書きかえてみてください。

```
...
inum = (int)dnum;
...
```

変換する型を指定します

実行結果はさきほどと同じようになります。ただし、今度は小さなサイズの型に変換することをはっきりとコード中に書きあらわすことができました。キャスト演算子を使えば、型を変換することを明示的に記述することができるのです。

重要

キャスト演算子は型を変換する。

160.5
dnum
double 型
inum = (int) dnum;
代入の際に小さいサイズの
型に変換される
160
160
inum
int 型

図 4-10 小さなサイズの型への代入
小さなサイズの型の変数に、大きなサイズの型の値を代入すると、
値の一部が失われることがあります。

キャスト演算子の使用

キャスト演算子は、Sample8のように大きなサイズの型に変換する場合にも使うことができます。

```
dnum = (double)inum;
```

double型に変換します

ただし、Sample8をこのように書きかえても、実行結果は同じです。

異なる型どうしで演算する

代入をする場合の型変換について、理解することができたでしょうか？ 次に、たし算、ひき算、かけ算、わり算といった四則演算などを行う演算子とオペランド型の関係についてみていきましょう。次の例をみてください。

Sample10.cpp ▶ 異なる型の演算をする

```
#include <iostream>
using namespace std;

int main()
{
   int d = 2;
   const double pi = 3.14;

   cout << "直径が" << d << "センチの円の¥n";
   cout << "円周は" << d*pi << "センチです。¥n";

   return 0;
}
```

int型のdがdouble型に変換されて演算されます

Sample10の実行画面

```
直径が2センチの円の
円周は6.28センチです。
```

ここでは、int型のdの値とdouble型のpiの値のかけ算をしています。C++では一般的に、演算子に異なる型のオペランドを記述した場合、

一方のオペランドを大きなサイズのほうに型変換してから演算を行う

というきまりになっています。つまり、ここではint型のdの値の「2」がdouble型の数値（2.0）に変換されてから、かけ算が行われるのです。得られる式の値もdouble型の値となります。

図4-11 異なる型の演算
型が異なる場合、大きなサイズの型に変換されてから演算が行われます。得られる値も大きなサイズの型になります。

同じ型どうしで演算する

それでは、同じ型どうしで演算を行う場合はどうなるのでしょうか？ この場合は同じ型どうしで演算が行われ、結果もその型の値となります。しかし、この演算が予想外の結果となってしまうコードもあります。次の例をみてください。

Lesson 4

Sample11.cpp ▶ 同じ型の演算をする

```
#include <iostream>
using namespace std;

int main()
{
   int num1 = 5;
   int num2 = 4;
   double div;

   div = num1/num2;   // 5÷4を計算しているつもりなのですが・・・

   cout << "5/4は" << div << "です。¥n";

   return 0;
}
```

Sample11の実行画面

```
5/4は1です。   // 思ったとおりの答えになっていません
```

このコードは、int型の変数num1とnum2を使って、5÷4の結果をdouble型の変数divに代入しようとしたものです。「1.25」というdouble型が出力されることを期待して、このように記述してみたのです。

しかし、このnum1とnum2はint型であるため、「5/4」が、「1」というint型の値として計算されます。その結果、出力は「1」という値になってしまっています。

図4-12 同じ型の演算
2つのオペランドがともにint型の場合は、結果もint型となります。

このコードで「1.25」という出力を得るためには、変数num1かnum2の少なくとも一方をdouble型に変換する必要があります。わり算の部分をキャスト演算子を使って、次のように書きかえます。

```
...
div = (double)num1/(double)num2;
...
```

キャスト演算子を使います

すると、コードを変更した場合の実行画面は次のようになります。

変更後のSample11の実行画面

```
5/4は1.25です。
```

今度は意図どおりに出力されました

このようにコードを書きかえることによって、double型の演算が行われるようになります。結果もdouble型となり、「1.25」という答えを出力することができます。演算をする場合は、オペランドの型に十分注意しておく必要があります。

図4-13 double 型の演算

Sample11 で double 型の結果を得たい場合は、少なくとも一方のオペランドを、キャスト演算子で型変換します。

4.5 レッスンのまとめ

この章では、次のようなことを学びました。

- 演算子は、オペランドと組みあわせて式をつくります。
- インクリメント・デクリメント演算子を使うと、変数の値を１加算または減算できます。
- 複合的な代入演算子を使うと、四則演算と代入演算を組みあわせた処理を行うことができます。
- sizeof 演算子で、型や式のサイズを知ることができます。
- 異なる基本型どうしをオペランドとした場合、型変換が行われる場合があります。
- 値が小さなサイズの型に変換された場合、値の一部が失われる場合があります。
- キャスト演算子を使うと、明示的に型を変換できます。
- 代入の際に、型が変換される場合があります。
- 四則演算などの際に、型が変換される場合があります。

演算子を使うと、四則演算をはじめとするいろいろな処理をコンパクトに記述できるようになります。ここでは、すべての演算子の扱いをみることはできません。しかし、これからとりあげるコードでは、さまざまな演算子が登場してくることになります。わからない演算子があった場合は、この章に戻って復習するようにしてください。

練習

1. 次の計算結果を出力するコードを記述してください。

0-4
3.14×2
5÷3
30÷7のあまりの数
(7+32)÷5

2. 次のように高さと底辺を入力して、三角形の面積を出力するコードを記述してください。
（ヒント：三角形の面積＝（高さ×底辺）÷2）

```
三角形の高さを入力してください。
3 ↵
三角形の底辺を入力してください。
5 ↵
三角形の面積は7.5です。
```

3. キーボードから5科目のテストの点数を入力して、次のように合計値と平均値を出力するコードを記述してください。

```
科目1の点数を入力してください。
52 ↵
科目2の点数を入力してください。
68 ↵
科目3の点数を入力してください。
75 ↵
科目4の点数を入力してください。
83 ↵
科目5の点数を入力してください。
36 ↵
5科目の合計点は314点です。
5科目の平均点は62.8点です。
```

Lesson 5

場合に応じた処理

ここまでのプログラムでは、コード内の文が、1文ずつ順序よく処理されていました。しかし、さらに複雑な処理をしたい場合には、順番に処理するだけでは対応できないことがあります。C++では複数の文をまとめて、処理をコントロールする方法があります。この章では、特定の状況に応じて処理をコントロールする文を学びましょう。

Check Point!

- 条件
- 関係演算子
- 条件判断文
- if文
- if～else文
- if～else if～else文
- 論理演算子
- switch文

5.1 関係演算子と条件

条件のしくみを知る

私たちは、日常生活で次のような状況に出会う場合がありますね。

学校の成績がよかったら・・・
→　友達と旅行に行く
学校の成績が悪かったら・・・
→　もう一度勉強する

C++でも、このように「場合に応じた処理」を行うことができます。この章では、さまざまな状況に応じた複雑な処理を行う方法を学びましょう。

C++でさまざまな状況をあらわすためには、**条件**（condition）という概念を使います。

たとえば上の例では、

よい成績である

ことが、「条件」にあたります。

もちろん、実際のC++のコードでは、このように日本語で条件を記述するわけではありません。まず、第4章で学んだ式を思い出してください。第4章では、式が評価されて値をもつことを学びました。このような式のうち、

真（true）
偽（false）

という2つの値のどちらかであらわされるものを、C++では「条件」と呼びます。trueまたはfalseとは、その条件が「正しい」または「正しくない」ということをあらわす値です。

たとえば、「よい成績である」という条件を考えてみると、条件がtrueまたは

falseになる場合とは、次のようなことをいうわけです。

成績が80点以上だった場合 → よい成績であるから条件はtrue
成績が80点未満だった場合 → よい成績でないから条件はfalse

条件を記述する

条件というものが、なんとなくわかったところで、条件をC++の式であらわしてみましょう。私たちは、1より3が大きいことを、

3 > 1

という不等式であらわすことがあります。たしかに、3は1より大きい数値なので、この不等式は「正しい」といえます。一方、この不等式はどうでしょうか。

3 < 1

この式は「正しくない」ということができます。C++でも、>のような記号を使うことができ、上の式はtrue、下の式はfalseであると評価されます。つまり、3>1や3<1という式は、C++の条件ということができるのです。

図5-1 条件
関係演算子を使って「条件」を記述することができます。条件は、trueまたはfalseという値をもちます。

条件をつくるために使う>記号などは、**関係演算子**（relational operator）と呼ばれています。表5-1に、いろいろな関係演算子をまとめました。

表5-1をみるとわかるように、> の場合は「右辺より左辺が大きい場合にtrue」となるので、3>1はtrueとなります。これ以外の場合、たとえば1>3はfalseとなります。

表5-1：関係演算子

演算子	式がtrueとなる場合
==	右辺が左辺に等しい
!=	右辺が左辺に等しくない
>	右辺より左辺が大きい
>=	右辺より左辺が大きいか等しい
<	右辺より左辺が小さい
<=	右辺より左辺が小さいか等しい

重要

関係演算子を使って条件を記述する。

true と false

trueとfalseはbool型（表3-1）の値で、**論理リテラル**と呼ばれています。bool型の値は、比較的新しくC++にとりいれられたものです。そこで、trueとfalseは整数型の値に変換される場合もあります。この場合は

```
true   →  1
false  →  0
```

となります。

また、逆に、整数型の値がbool型の値に変換される場合もあります。この場合は、

```
0以外の整数  → true
0            → false
```

と変換されることになります。

重要

整数型の値に変換されるとき、trueは1、falseは0として扱われる。

関係演算子を使う

それでは、関係演算子を使って、いくつか条件を記述してみましょう。

```
5 > 3
5 < 3
a == 6
a != 6
```

- 5 > 3：この条件の評価はtrueです
- 5 < 3：この条件の評価はfalseです
- a == 6：この条件の評価は変数aの値によって異なります
- a != 6：この条件の評価は変数aの値によって異なります

「5 > 3」という条件は、3よりも5が大きいので、式の値はtrueになることがわかりますね。

また、「5 < 3」という条件は、3よりも5が大きいので、式の値はfalseになります。

条件の中に変数を使うこともできます。たとえば、上の「a == 6」という条件は、変数aの値が6であった場合、trueになります。一方、変数aの内容が3や10であった場合はfalseになります。このように、そのときの変数の値によって条件があらわす値が異なるわけです。

同様に、「a != 6」はaが6でない値のときにtrueとなる条件となっています。

なお、!=や==は2文字で1つの演算子ですから、!と=の間に空白を入力したりしてはいけません。

図5-2 **変数と条件**
変数を条件に使った場合は、変数の値によって評価が異なります。

ところで、= 演算子が代入演算子と呼ばれていたことを思い出してください（第4章）。かたちは似ていますが、== は異なる種類の演算子（関係演算子）です。この2つの演算子は、実際にコードを書く際にたいへんまちがえやすい演算子です。よく注意して入力するようにしてください。

重要

= （代入演算子）と == （関係演算子）をまちがえないこと。

5.2 if文

if文のしくみを知る

それでは、この章の目的である、さまざまな状況に応じた処理を行ってみましょう。

C++では、状況に応じた処理を行う場合、

「条件」の値（trueまたはfalse）に応じて処理を行う

というスタイルの文を記述します。このような文を**条件判断文**（conditional statement）といいます。まずはじめに、条件判断文のひとつとして、**if文**（if statement）という構文を学びましょう。if文は、条件がtrueの場合に、指定した文を処理するという構文です。

構文 **if文**

```
if (条件)
   文;
```

条件がtrueのときに処理されます

図5-3 if文
if文は、条件がtrueだった場合に、指定した文を処理します。falseの場合には、指定した文を処理しないで次の処理にうつります。

たとえば、最初にあげた例をif文にあてはめてみると、次のような感じのコードになります。

```
if (よい成績をとった)
    旅行に行く
```

if文を記述することによって、条件（「よい成績をとった」）がtrueであった場合、「旅行に行く」という処理を行うのです。悪い成績だった場合には、「旅行に行く」という処理は行われません。

それでは、実際にコードを入力して、if文を実行してみることにしましょう。

Sample1.cpp ▶ if文を使う

```
#include <iostream>
using namespace std;

int main()
{
    int res;

    cout << "整数を入力してください。¥n";

    cin >> res;

    if (res == 1)
        cout << "1が入力されました。¥n";

    cout << "処理を終了します。¥n";

    return 0;
}
```

① 変数resにキーボードからの入力を読み込みます（cin >> res;）

② 1が入力された場合、この条件がtrueとなり・・・（if (res == 1)）

③ この文が処理されます（cout << "1が入力されました。¥n";）

Sample1の実行画面

```
整数を入力してください。
1 ↵
1が入力されました。
処理を終了します。
```

1が入力されたので・・・（1 ↵）

③の部分が処理されました（1が入力されました。）

Sample1では条件res==1がtrueであれば、③の部分が処理されます。falseの場合は、③の部分は処理されません。

したがって、ユーザーがプログラムを実行して1を入力した場合は、条件res == 1がtrueとなるので、③の部分が処理されます。そのため、上のように画面に出力されるのです。

それでは、ユーザーが1以外の文字を入力したらどうなるでしょうか。

Sample1の実行画面

```
整数を入力してください。
10 ↵          ← 1以外が入力されたので・・・
処理を終了します。 ← ③の部分は処理されません
```


今度は、res == 1という条件がfalseになるため、③の部分は処理されません。したがって、実行したときの画面は上のようになります。このように、if文を使うと、条件がtrueのときにだけ、処理を行うことができます。

```
if (res == 1)  ── true ──┐
  │                      ↓
false   cout << "1が入力されました。¥n"; ─┐
  ↓                                       │
cout << "処理を終了します。¥n"; ←──────────┘
```

図5-4 if文の流れ

if文で複数の文を処理する

Sample1では条件がtrueだった場合に、1つだけの文からなる、かんたんな処理を行いました。if文では条件がtrueのとき、複数の文を処理することもできます。このためには、{ }でブロックをつくり、複数の文をまとめます。すると、ブロッ

ク中は原則どおり、1文ずつ順に処理されます。

構文　複数の文を処理するif文

```
if(条件){
    文1;
    文2;
    ...
}
```

条件がtrueのとき、順番に処理されます

if
条件
true
false
文1
文2

図5-5　複数の文をif文中で処理する
if文では、{}ブロック内の複数の文を処理することができます。

具体的に例をみてみましょう。

Sample2.cpp ▶ 複数の文を処理するif文を使う

```
#include <iostream>
using namespace std;

int main()
{
   int res;

   cout << "整数を入力してください。¥n";

   cin >> res;

   if (res == 1){
```

1が入力された場合(条件がtrueの場合)・・・

```
        cout << "1が入力されました。¥n";
        cout << "1を選択しました。¥n";
    }

    cout << "処理を終了します。¥n";

    return 0;
}
```

ブロック内が順に処理されます

Sample2の実行画面その1

```
整数を入力してください。
1 ↵
1が入力されました。
1を選択しました。
処理を終了します。
```

ブロック内が順に処理されています

ユーザーが1を入力した場合は条件がtrueとなるので、ブロック内の処理が順に行われ、2行文の文字列が出力されます。もし1以外の数値を入力した場合は、ブロック内の処理は実行されず、次のようになります。

Sample2の実行画面その2

```
整数を入力してください。
10 ↵
処理を終了します。
```

ブロック内の処理は行われません

さきほどの結果とくらべると、行われていない処理があるのがわかりますね。

```
if (res == 1){          true
    cout << "1が入力されました。¥n";
    cout << "1を選択しました。¥n";
}                       false
cout << "処理を終了します。¥n";
```

図5-6 複数の文をif文中で処理する

ブロックにしないと？

次のコードはSample2に似ていますが、実行するとどうなるでしょうか？

```
#include <iostream>
using namespace std;

int main()
{
   int res;

   cout << "整数を入力してください。¥n";

   cin >> res;

   if (res == 1)
      cout << "1が入力されました。¥n";
      cout << "1を選択しました。¥n";

   cout << "処理を終了します。¥n";

   return 0;
}
```

この文(①)だけがif文の内容になっています

この文(②)はif文の外の処理になっています

```
整数を入力してください。
2 ↵
1を選択しました。 ● おかしな出力になっています
処理を終了します。
```

画面をみると、意図しない処理が行われているのがわかります。これは、複数の文を処理させようとして記述したにもかかわらず、{}でかこむことを忘れてしまったからです。ブロックがないため、if文の内容は①だけであるとされてしまったのです。

このようなことを防ぐためには、どこがif文の構文であるのか、わかりやすいように字下げ（インデント）を行ったり、1文であってもブロックでかこむなどして、読みやすいコードを心がけるべきです。

重要 ブロック内は、字下げ（インデント）を使って読みやすくする。

セミコロンに注意

if文では、セミコロンの位置に注意してください。通常if文の1行目に条件を記述して改行しますが、この行にはセミコロンは必要ありません。

```
if(res == 1)                        ● この行にはセミコロンをつけません
    cout << "1が入力されました。¥n"; ● この行にはセミコロンをつけます
```

なお、まちがって1行目にセミコロンをつけてしまっても、コンパイラはエラーを表示しません。実行したときにうまく動作しなくなってしまうので、注意しておくようにしましょう。

5.3 if ～ else 文

if ～ else 文のしくみを知る

さて、5.2節のif文では、条件がtrueの場合にだけ、特定の処理をしていましたね。さらに、if文のバリエーションとして、条件がfalseの場合に、指示した文を実行させる構文もあります。これが**if～else文**です。

構文 **if~else文**

```
if(条件)
    文1;
else
    文2;
```

この構文では、条件がtrueの場合に文1を処理し、falseの場合に文2を処理します。

この章の最初の例にたとえてみると、

```
if(よい成績をとった)
    旅行に行く
else
    もう一度勉強する
```

といった具合になります。今度は「よい成績をとった」という条件がfalseの場合にも、特定の処理（「もう一度勉強する」）を行うことができます。

また、if～else文でも{ }でかこんで複数の文を処理させることができます。この構文は次のようになります。

構文 **複数の文を処理するif~else文**

```
if(条件){
    文1;
    文2;      条件がtrueのとき、順番に処理されます
    ...
}
else{
    文3;
    文4;      条件がfalseのとき、順番に処理されます
    ...
}
```


この構文では、条件がtrueの場合に文1、文2・・・と順に処理し、falseの場合に文3、文4・・・と順に処理します。

図5-7 if～else文
if～else文では、条件がtrueだった場合とfalseだった場合に、異なる処理を行うことができます。ブロック内の複数文も処理できます。

それでは、if～else文をためしてみるために、次のコードを入力してみましょう。

Sample3.cpp ▶ if～else 文を使う

```
#include <iostream>
using namespace std;

int main()
{
    int res;

    cout << "整数を入力してください。¥n";

    cin >> res;

    if (res == 1){
        cout << "1が入力されました。¥n";
    }
    else{
        cout << "1以外が入力されました。¥n";
    }

    return 0;
}
```

① 1が入力された場合（条件がtrueの場合）に処理されます

② 1以外が入力された場合（条件がfalseの場合）に処理されます

Sample3の実行画面その1

```
整数を入力してください。
1 ↵
1が入力されました。
```

Sample3の実行画面その2

```
整数を入力してください。
10 ↵
1以外が入力されました。
```

ユーザーが1を入力した場合と10を入力した場合の、2とおりの画面を示しました。1を入力した場合は、いままで同様に①が処理されますが、それ以外の場合は②が処理されます。if～else文では、このように複雑な処理を行うことができます。

```
        true
false ┌ if (res == 1){ ─┐
      │                 ↓
      │     cout << "1が入力されました。¥n"; ─┐
      │ }                                    │
      │                                      │
      │ else{                                │
      │                                      │
      └→    cout << "1以外が入力されました。¥n";  │
        }               │                    │
                        ↓                    ↓
```

図5-8 **if～else文の流れ**

複文

複数の文をまとめたブロックは**複文**と呼ばれます。if文では単文のかわりに、複文を記述できます。C++では単文が記述されているところであれば、どこでも複文を記述することができるようになっています。

5.4 複数の条件を判断する

if ～ else if ～ elseのしくみを知る

if文では、2つ以上の条件を判断させて処理するバリエーションをつくることもできます。これがif～else if～elseです。この構文を使えば、2つ以上の条件に応じた処理ができるようになります。

構文 if～else if～else

```
if(条件1){
    文1;
    文2;
    ...
}
else if(条件2){
    文3;
    文4;
    ...
}
else if(条件3){
    ...
}
else{
    ...
}
```


この構文では条件1を判断し、trueだった場合は文1、文2・・・の処理を行います。もしfalseだった場合は、条件2を判断して、文3、文4・・・の処理を行います。このように次々と条件を判断していき、どの条件もfalseだった場合は、最後のelse以下の文が処理されます。

たとえば、

```
if(成績は「優」だった)
    外国に旅行に行く
else if(成績は「可」だった)
    国内旅行に行く
else if(成績は「不可」だった)
    もう一度勉強する
```

といった具合です。かなり複雑な処理ができることがわかりますね。

else ifの条件はいくつでも設定でき、**最後のelseは省略することも可能です**。最後のelseを省略し、かつどの条件にもあてはまらなかった場合、このコードの中で実行される文は存在しないことになります。

図5-9 if～else if～else
if～else if ～elseでは、複数の条件に応じた処理ができます。

このしくみを使うと、複数の条件に応じた処理をすることができます。
それでは、コードを記述してみることにしましょう。

Sample4.cpp ▶ if～else if～elseを使う

```
#include <iostream>
using namespace std;

int main()
{
    int res;

    cout << "整数を入力してください。¥n";

    cin >> res;

    if (res == 1){
        cout << "1が入力されました。¥n";
    }
    else if(res == 2){
        cout << "2が入力されました。¥n";
    }
    else{
        cout << "1か2を入力してください。¥n";
    }

    return 0;
}
```

① 1が入力された場合に処理されます

② 2が入力された場合に処理されます

③ 1・2以外が入力された場合に処理されます

Sample4の実行画面その1

```
整数を入力してください。
1 ↵
1が入力されました。
```

Sample4の実行画面その2

```
整数を入力してください。
2 ↵
2が入力されました。
```

Sample4の実行画面その3

```
整数を入力してください。
3 ↵
1か2を入力してください。
```

1を入力した場合、最初の条件はtrueになるので、①が処理され、ほかの部分は処理されません。

2を入力した場合、最初の条件がfalseとなるため、次の条件を判断します。2番目の条件はtrueとなるので、今度は②が処理されます。

1や2以外を入力した場合（1・2番目の条件ともfalseの場合）、必ず③が処理されます。このように、if～else if～elseを使うと、いくつもの条件を判断していって、複雑な処理を行うことができるのです。

図5-10 if～else if ～elseの流れ

5.5 switch文

switch文のしくみを知る

C++には、if文と同じように、条件によって処理をコントロールできる**switch文**（switch statement）という構文があります。switch文は次のようになっています。

構文　switch文

```
switch(式){
   case 値1:
      文1;
      ...
      break;
   case 値2:
      文2;
      ...
      break;
   default:
      文D;
      ...
      break;
}
```


switch文では、switch文内の式がcaseのあとの値と一致すれば、そのあとの文から「break」までの文を実行します。もしどれにもあてはまらなければ、「default:」以下の文を実行します。「default:」は、省略することも可能です。

switch文をたとえてみると、このような文になります。

```
switch(成績){
   case 優:
      外国旅行に行く
      break;
```

```
    case 可:
        国内旅行に行く
        break;
    case 不可:
        もう一度勉強する
        break;
}
```

このswitch文では、成績によっていろいろな処理を行っています。if～else if～elseと同じ処理になっていることがわかるでしょう。switch文を使うと、if～else if～elseをかんたんに記述できる場合があります。

Lesson 5

重要

switch文を使うと、if～else if～elseをかんたんに記述できる場合がある。

switch
case1:
break;
文
case2:
break;
文
default:
文

図5-11 switch文
switch文を使っても、複数の条件に応じた処理ができます。

swtich文を使った例をみてみることにしましょう。

Sample5.cpp ▶ switch文を使う

```
#include <iostream>
using namespace std;

int main()
{
   int res;

   cout << "整数を入力してください。¥n";

   cin >> res;

   switch(res){
      case 1:
         cout << "1が入力されました。¥n";
         break;
      case 2:
         cout << "2が入力されました。¥n";
         break;
      default:
         cout << "1か2を入力してください。¥n";
         break;
   }

   return 0;
}
```

1が入力された場合に処理されます

2が入力された場合に処理されます

1・2以外が入力された場合に処理されます

このコードは、変数resの値を判断し、Sample4のif〜else if 〜elseと、まったく同じ処理をしています。実行結果も同じです。

switch文では、いくつも条件がある複雑なif〜else if〜elseをシンプルに書きあらわすことができる場合があるのです。ただし、switch文の場合、値を判断する式（ここでは変数res）は、整数型でなければなりません。

break文が抜けていると？

switch文を扱うにあたっては、いくつかの注意があります。下のコードをみてください。このコードは、Sample5のコードからbreak文をはぶいたものです。

```
#include <iostream>
using namespace std;

int main()
{
    int res;

    cout << "整数を入力してください。¥n";

    cin >> res;

    switch(res){
        case 1:
            cout << "1が入力されました。¥n";
        case 2:
            cout << "2が入力されました。¥n";
        default:
            cout << "1か2を入力してください。¥n";
    }

    return 0;
}
```

break文が抜けているswitch文です

これを実行すると、画面は次のようになってしまいます。

```
数値を入力してください。
1 ↵
1が入力されました。
2が入力されました。
1か2を入力してください。
```

おかしな出力となってしまいます

このコードでは、1を入力したときに、case 1:以降の文がすべて実行されてしまっていて、おかしなことになっています。

そもそもbreakという文は、

文の流れを強制的に切る

という役割をもっています。switch文では、break文が出てくるか、ブロックが終了するまで、ブロックの中の文が順次処理されるため、正しい位置にbreak文を入れないとおかしな結果となってしまいます。

break文を書き忘れたり、位置をまちがえたりしても、コンパイラはエラーを表示しませんので、注意してください。break文については、次の章でもう一度学ぶことにしましょう。

重要

switch文の中のbreak文の位置に気をつける。

5.6 論理演算子

論理演算子のしくみを知る

これまで、いろいろな条件を使った条件判断文を記述してきました。このような文の中で、もっと複雑な条件を書ければ便利な場合があります。たとえば、次のような場合を考えてみてください。

成績が「優」であり、かつ、お金があったら・・・
→　海外旅行に行く

この場合の条件にあたる部分は、5.1節でとりあげた例よりも、もう少し複雑な場合をあらわしています。このような複雑な条件をC++で記述したい場合には、**論理演算子**（logical operator）という演算子を使います。論理演算子は、

条件をさらに評価して、trueまたはfalseの値を得る

という役割をもっているものです。
たとえば、上の条件を論理演算子を使って記述すると、次のようになります。

(成績が「優」である) && (お金がある)

&&演算子は、左辺と右辺がともにtrueの場合に、全体の値をtrueとする論理演算子です。この場合は、「成績が優であり」かつ「お金がある」場合に、この条件はtrueとなります。どちらか一方でも成立しない場合は、全体の条件はfalseとなり、成立しないことになります。
論理演算子は、次の表のように評価されることになっています。

表5-2：論理演算子

<table>
<tr><th rowspan="2">演算子</th><th rowspan="2">trueになる場合</th><th colspan="3">評価</th></tr>
<tr><th>左</th><th>右</th><th>全体</th></tr>
<tr><td rowspan="4">&&</td><td rowspan="4">左辺・右辺がともにtrueの場合
左辺：true　右辺：true</td><td>false</td><td>false</td><td>false</td></tr>
<tr><td>false</td><td>true</td><td>false</td></tr>
<tr><td>true</td><td>false</td><td>false</td></tr>
<tr><td>true</td><td>true</td><td>true</td></tr>
<tr><td rowspan="5">||</td><td rowspan="5">左辺・右辺のどちらかがtrueの場合
左辺：true　右辺：true</td><th>左</th><th>右</th><th>全体</th></tr>
<tr><td>false</td><td>false</td><td>false</td></tr>
<tr><td>false</td><td>true</td><td>true</td></tr>
<tr><td>true</td><td>false</td><td>true</td></tr>
<tr><td>true</td><td>true</td><td>true</td></tr>
<tr><td rowspan="3">!</td><td rowspan="3">右辺がfalseの場合
右辺：true</td><th></th><th>右</th><th>全体</th></tr>
<tr><td></td><td>false</td><td>true</td></tr>
<tr><td></td><td>true</td><td>false</td></tr>
</table>

それでは、論理演算子を使ったコードを、具体的にみてみましょう。

```
5>3 && 3==4
a==6 || a>=12
!(a==6)
```

① この条件はfalseです

② この条件は変数aの値が6または12以上のときにtrueになります

③ この条件は変数aの値が6ではないときにtrueとなります

&&演算子を使った式は、左辺・右辺がともにtrueとなる場合のみ全体がtrueとなるのでした。したがって、条件①の値はfalseです。

||演算子を使った式は、左辺・右辺のどちらかがtrueであれば、全体の式がtrueとなります。したがって、条件②では、変数aに入っている値が6や13だった場合はtrueになります。また、aが5だった場合はfalseとなります。

!演算子はオペランドを1つとる単項演算子で、オペランドがfalseのときにtrueとなります。条件③では変数aが6ではない場合にtrueとなるわけです。

重要

論理演算子を使うと、条件を組みあわせて複雑な条件をつくることができる。

図5-12 **論理演算子**
論理演算子は、trueかfalseの値を演算します。

Lesson 5

複雑な条件判断をする

さて、いままで学んだif文などの中で論理演算子を使えば、より複雑な条件を判断する処理ができるようになります。

さっそく、論理演算子を使ってみましょう。

Sample6.cpp ▶ 論理演算子で条件を記述する

```
#include <iostream>
using namespace std;

int main()
{
   char res;
```

```
    cout << "あなたは男性ですか？¥n";
    cout << "YまたはNを入力してください。¥n";

    cin >> res;

    if (res == 'Y' || res == 'y'){
       cout << "あなたは男性ですね。¥n";
    }
    else if(res == 'N' || res == 'n'){
       cout << "あなたは女性ですね。¥n";
    }
    else{
       cout << "YかNを入力してください。¥n";
    }

    return 0;
}
```

Yまたはyが入力された場合に処理されます

Nまたはnが入力された場合に処理されます

Y、y、N、n以外が入力された場合に処理されます

Sample6の実行画面その1

```
あなたは男性ですか？
YまたはNを入力してください。
Y ↵
あなたは男性ですね。
```

Sample6の実行画面その2

```
あなたは男性ですか？
YまたはNを入力してください。
n ↵
あなたは女性ですね。
```

Sample6では、キーボードから入力した文字にしたがって処理を行っています。文字には、Yとyのように大文字と小文字がありますが、ここでは、大文字・小文字の区別なく処理をしたいと考えました。そこで、Sample6では、if文の条件に論理演算子||を使ってみたのです。なお、文字は''でかこむことを思い出してください。

||を使ってif文を記述すれば、Yまたはyを入力したときに、同じ処理を行う

ことができます。

ビット単位の論理演算子

C++には、2進数で数値をあらわした場合のケタ（ビット）どうしの演算を行う、「ビット単位の論理演算子」という演算子が用意されています。

ビット単位の論理演算子とは、

2進数であらわした1つまたは2つの数値のそれぞれのケタについて、0か1のどちらかを返す

という演算を行うものです。

たとえば、ビット論理積演算子の「&」は、2つの数値のケタがどちらも1だった場合は1、それ以外は0とする演算を行います。次の例をみてください。

```
  5   0000000000000101
&12   0000000000001100   ● 1 & 0 → 0と評価します
-----------------------
  4   0000000000000100
```

short int型の数値を使って「5 & 12」という演算を行うと、上のようになります。計算結果の数値は4になります。

これらの種類の演算子の用法について、下の表にまとめておきます。ビット単位の論理演算子は、整数型の値を演算する演算子です。trueまたはfalseに対する演算を行う論理演算子とは異なるものなので、注意しておきましょう。

表：ビット単位の論理演算子

演算子	1となる場合	評価		
		左	右	全体
&	左辺・右辺のビットがともに1の場合 （図　左辺：1　右辺：1）	0	0	0
		0	1	0
		1	0	0
		1	1	1

演算子	1となる場合	評価		
		左	右	全体
|	左辺・右辺のビットのどちらかが1の場合	0	0	0
		0	1	1
		1	0	1
		1	1	1
^	左辺・右辺のビットが異なる場合	左	右	全体
		0	0	0
		0	1	1
		1	0	1
		1	1	0
~	右辺のビットが0の場合		右	全体
			0	1
			1	0

条件演算子のしくみを知る

これまで、複雑な条件判断を行う方法をみてきました。一方、かんたんな条件判断の場合は、if文を使わなくても、**条件演算子**(conditional opetrator)の「?:」を使って書くこともできます。次のようなコードをみてください。

```
#include <iostream>
using namespace std;

int main()
{
   int res;
   char ans;

   cout << "何番目のコースにしますか?¥n";
   cout << "整数を入力してください。¥n";
```

```
    cin >> res;

    if(res == 1)
       ans='A';
    else
       ans='B';

    cout << ans << "コースを選択しました。¥n";

    return 0;
}
```

if文を使った条件判断をしています

このコードは、res==1がtrueであるとき、変数ansに文字Aを代入し、それ以外の場合はBを代入する処理を、if文を使って記述したものです。このようなかんたんな条件判断は、条件演算子?:を使って次のように書きかえることができます。

Sample7.cpp ▶ 条件演算子を使う

```
#include <iostream>
using namespace std;

int main()
{
   int res;
   char ans;

   cout << "何番目のコースにしますか?¥n";
   cout << "整数を入力してください。¥n";

   cin >> res;

   ans = (res==1) ? 'A' : 'B';

   cout << ans << "コースを選択しました。¥n";

   return 0;
}
```

if文を条件演算子で書きかえました

Lesson 5

Sample7の実行画面

```
何番目のコースにしますか？
整数を入力してください。
1 ↵
Aコースを選択しました。
```

if文よりもかんたんに記述できているのがわかりますね。

条件演算子？:の使いかたをまとめておきましょう。

構文 **条件演算子**

```
条件 ? trueのときの式1 : falseのときの式2
```

条件演算子は、3つのオペランドをとる演算子です。全体の式の値は、条件がtrueのときに式1の値、falseの場合には式2の値となります。

Sample7の式の値は、res==1がtrueであるときにA、それ以外の場合はBとなります。つまり、どちらかの値が変数ansに代入されるというわけです。

図5-13 条件演算子
条件演算子は、先頭に記述した条件の値によって、式の値が決まります。

重要
条件演算子を使うと、かんたんな条件に応じた処理を記述できる。

5.7 レッスンのまとめ

この章では、次のようなことを学びました。

- 関係演算子を使って、条件を作成できます。
- if 文を使って、条件に応じた処理をできます。
- if 文のバリエーションを使って、いろいろな条件に応じた処理ができます。
- switch 文を使って、式の値に応じた処理ができます。
- 論理演算子を使って、複雑な条件を作成できます。
- 条件演算子 ?: を使って、かんたんな条件に応じた処理を記述できます。

if文やswitch文を使うと、条件に応じた処理をして、柔軟なコードを記述することができます。さまざまな状況に応じたコードが書けるようになったことでしょう。次の章で、繰り返しを行う構文を学ぶと、より強力なコードを記述できるようになります。

練習

1. キーボードから整数値を入力させ、場合に応じて次のようなメッセージを出力するコードを記述してください。

値が偶数だった場合 ――――――「○は偶数です。」
値が奇数だった場合 ――――――「○は奇数です。」
(ただし○は入力した整数)

```
整数を入力してください。
1 ↵
1は奇数です。
```

2. キーボードから2つの整数値を入力させ、場合に応じて次のようなメッセージを出力するコードを記述してください。

値が同じ場合 ――――「2つの数は同じ値です。」
それ以外の場合 ―――「○より×のほうが大きい値です。」
(ただし○、×は入力した整数。○<×)

```
2つの整数を入力してください。
1 ↵
3 ↵
1より3のほうが大きい値です。
```

3. キーボードから1から5までの5段階の成績を入力させ、場合に応じて次のようなメッセージを出力するコード記述してください。

成績	メッセージ
1	成績は1です。もっとがんばりましょう。
2	成績は2です。もう少しがんばりましょう。
3	成績は3です。さらに上をめざしましょう。
4	成績は4です。たいへんよくできました。
5	成績は5です。たいへん優秀です。

```
成績を入力してください。
3 ↵
成績は3です。さらに上をめざしましょう。
```

Lesson 6

何度も繰り返す

第5章では、条件に応じて処理をコントロールする文を学びました。C++ では、ほかにも文をコントロールする機能が用意されています。この機能は「繰り返し文（ループ文）」と呼ばれています。繰り返し文を使うと、同じ処理を何度も繰り返すことができるようになります。この章では、繰り返し文について学びましょう。

Check Point!

- 繰り返し文
- for文
- while文
- do～while文
- 文のネスト
- break文
- continue文

6.1 for文

for文のしくみを知る

第5章では、条件の値にしたがって処理する文をコントロールする方法を学びました。C++ではほかにも、複雑な処理を行うことができます。たとえば、次のような状況を考えてみてください。

試験に合格するまでは・・・
→　試験を受け続ける

私たちは、日常生活でも一種の「繰り返しの処理」を行っていることがあります。朝起きる、歯をみがく、朝食を食べ、学校に行く・・・、私たちの生活はこういうことの繰り返しです。

C++では、このような処理を**繰り返し文**（ループ文：loop statement）と呼ばれる構文で記述することができます。C++の繰り返し文には、for文・while文・do～while文の3つのバリエーションがあります。

この章ではまず、**for文**（for statement）から順番に学んでいくことにしましょう。for文のスタイルを最初にみてください。

構文　for文

```
for(初期化の式1; 繰り返すかどうか調べる式2; 変化のための式3)
    文;
```

この文を繰り返し実行します

for文の処理のくわしい手順については、例を入力しながら学ぶことにしますので、ここではかたちだけをながめておきます。

またfor文では、if文と同じように、複数の文についても処理することができます。複数の文を繰り返したい場合は、if文のときと同様に{ }でかこみ、ブロックにします。

構文 **for文**

```
for(初期化の式1; 繰り返すかどうか調べる式2; 変化のための式3){
    文1;
    文2;        ブロック内の文を順に繰り返し処理します
    ・・・
}
```


図6-1 for文
for文を使うと、繰り返し処理を行うことができます。

for文でブロックを使うと、ブロック内の文1、文2・・・という処理を繰り返し行うことができるわけです。

それでは、実際にfor文を使ってみることにしましょう。

Sample1.cpp ▶ for文を使う

```
#include <iostream>
using namespace std;

int main()
{
    for(int i=1; i<=5; i++){          // 変数iを1つずつ増やしながら、i<=5がfalseになるまで・・・
        cout << "繰り返しています。¥n";   // この文が繰り返されます
    }

    cout << "繰り返しが終わりました。¥n";

    return 0;
}
```

Sample1の実行画面

```
繰り返しています。
繰り返しています。
繰り返しています。
繰り返しています。
繰り返しています。
繰り返しが終わりました。
```

for文では、繰り返す回数をカウントするために変数を用います。たとえば、このコードでは、変数iを使っています。そして、次のような手順で処理を行うことになります。

① 式1にしたがって、変数iを初期化する

↓

② 式2の条件がtrueであれば、ブロック内の文を処理してから、式3を処理する

↓

③ 式2の条件がfalseになるまで②を繰り返す

つまり、このfor文では変数iを1で初期化したあと、条件i <= 5がfalseになるまでi++を繰り返し、「繰り返しています。」と出力する文を処理するのです。

for文を理解するには、次のような状況を思いうかべてみるとわかりやすいかもしれません。

```
for(int i=1; i<=5; i++){
    試験を受ける
}
```

for文の処理では、変数iが1から5まで増えていく間に、試験を受けることを繰り返します。つまり、この場合には、試験を全部で5回繰り返して受けることになるわけです。

重要

for文を使うと、繰り返し処理が記述できる。

変数をループ内で使う

さて、Sample1では、繰り返しのたびに画面に文字が出力されるようにしていました。このとき、繰り返した回数を出力させることができたら便利です。そこで、次のコードを入力してみましょう。

Lesson 6

Sample2.cpp ▶ 繰り返し回数を出力する

```
#include <iostream>
using namespace std;

int main()
{
   for(int i=1; i<=5; i++){
      cout << i << "番目の繰り返しです。¥n";
   }

   cout <<"繰り返しが終わりました。¥n";

   return 0;
}
```

繰り返し処理の中で変数iを使っています

Sample2の実行画面

```
1番目の繰り返しです。
2番目の繰り返しです。
3番目の繰り返しです。
4番目の繰り返しです。
5番目の繰り返しです。
繰り返しが終わりました。
```

繰り返しのたびに値が増えます

繰り返し文の中では、回数のカウントに使っている変数iの値を出力することもできます。このコードを実行すると、ブロックの中で変数iの値が1つずつ増えながら繰り返されていることがよくわかるでしょう。何番目の繰り返しを処理したのかが一目でわかります。

たとえば、次の例をみてください。

```
for(int i=1; i<=5; i++){
    科目iの試験を受ける
}
```

この文は、科目1から5までの試験を全部で5回受けるという処理をあらわしています。複雑な処理を、かんたんなコードで記述できています。受ける科目が増えた場合にもすぐに対応できるでしょう。このように繰り返し文の中で変数を使うと、バリエーションにとんだプログラムが作成できるようになるのです。

なお、このfor文の中で宣言した変数iは、このfor文の中でだけ出力することができます。このiは、for文のブロックの外では使うことができないのです。もしブロックの外でもiを使いたい場合は、for文をはじめる前にiを宣言しておきます。

```
int i ;
for(i=1; i<5; i++){
    cout << i << "番目の繰り返しです。¥n";
}
    cout << i << "回繰り返しました。¥n";
```

for文の外にiを宣言すると・・・

for文の外でもiが使えます

重要

変数をfor文の繰り返しの中で使うと、繰り返し回数などが示せる。

for文を応用する

それでは、for文を応用したプログラムを、いくつか作成してみましょう。次のコードを入力してください。

Sample3.cpp ▶ 入力した数だけ*を出力させる

```
#include <iostream>
using namespace std;

int main()
{
    int num;

    cout << "いくつ*を出力しますか?¥n";

    cin >> num;

    for(int i=1; i<=num; i++){
        cout << '*';
    }
    cout <<'¥n';

    return 0;
}
```

数を入力させます

入力した数だけ*を繰り返し出力します

Sample3の実行画面

```
いくつ*を出力しますか?
10 ↵
**********
```

入力した数だけ*が出力されます

プログラムを実行すると、入力した数だけ*が出力されました。for文を使って、入力した数と同じだけ*の出力を繰り返して処理したのです。*の部分をほかの文字に変更すれば、いろいろな記号や文字を出力することができるでしょう。さまざまな文字でためしてみてください。

次に、1から入力した数までを順番にたし算した合計を求めるプログラムをつくってみることにしましょう。

Sample4.cpp ▶ 入力した数までの合計を求める

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int num;
    int sum = 0;

    cout << "いくつまでの合計を求めますか？¥n";

    cin >> num;  ● 数を入力させます

    for(int i=1; i<=num; i++){
        sum += i;  ● iが入力した数になるまでたし算を繰り返します
    }

    cout << "1から" << num << "までの合計値は" << sum << "です。¥n";

    return 0;
}
```

Sample4の実行画面

```
いくつまでの合計を求めますか？
10 ↵
1から10までの合計値は55です。
```

1から入力した数までの合計が求められます

ここでも同じように、入力した数まで繰り返し処理を行っていますね。

for文の中では、変数sumに変数iの値をたしていることに注目してください。変数iの値は1から1つずつ増えていきますから、この繰り返し処理で1から入力した数値までを合計した数を求めることができます。

sum		i		新しいsumの値	
0	+	1	=	1	1番目の繰り返し
1	+	2	=	3	2番目の繰り返し
3	+	3	=	6	3番目の繰り返し
6	+	4	=	10	4番目の繰り返し
				・・・	
45	+	10	=	55	10番目の繰り返し

6.2 while文

while文のしくみを知る

C++では、for文と同じように、指定した文を繰り返すことができる構文があります。そのひとつが**while文**（while statement）です。

構文 **while文**

```
while(条件){
    文;
    ...
}
```

条件がtrueの場合に・・・
ブロック内の文を順に繰り返し処理します

while文では、条件がtrueである限り、指定した文を何度でも繰り返し処理することができます。

この章のはじめの例にあてはめてみれば、while文とは次のような構文といえるでしょう。

```
while(試験に合格していない){
    試験を受ける
}
```

while文では、「試験に合格していない」という条件がfalseになるまで、試験を受ける処理を繰り返します。このwhile文では、処理をはじめる前に、試験にすでに合格していれば、試験を受ける処理は行われません。図をみながら、繰り返しの処理をつかんでみてください。

while
条件
true
文
false

図6-2 while文
while文を使うと、条件がfalseになるまで、繰り返し処理を行うことができます。

それでは、さっそくwhile文を使ったコードを記述してみましょう。

Sample5.cpp ▶ while文を使う

```
#include <iostream>
using namespace std;

int main()
{
    int i = 1;
    while(i <= 5){
        cout << i << "番目の繰り返しです。¥n";
        i++;
    }
    cout << "繰り返しが終わりました。¥n";

    return 0;
}
```

条件がtrueの場合に・・・

ブロック内の処理が順に繰り返されます

条件がfalseに近づくようにインクリメントしています

Sample5の実行画面

```
1番目の繰り返しです。
2番目の繰り返しです。
3番目の繰り返しです。
4番目の繰り返しです。
```

```
5番目の繰り返しです。
繰り返しが終わりました。
```

実は、このwhile文のコードの処理内容は、Sample2のfor文の処理とまったく同じです。このwhile文では、条件i<=5がfalseになるまで繰り返しを続けているからです。

このブロック内では条件がfalseに近づくように、変数iの値を増やしています。一般的に、繰り返し文では、繰り返しをするかしないかを判断するための条件が変化するようにしておかないと、永遠に繰り返し処理が行われることになってしまいます。たとえば、次のコードをみてください。


```
int i = 1;
while(i <= 5){
   cout << i << "番目の繰り返しです。¥n";
}
```

条件は決してfalseにならないので、ブロック内が永久に繰り返されてしまいます

このコードは、while文の条件の中に変数iの値を増やす「i++;」のような文を記述していないため、while文の条件はfalseになりません。このため、このようなプログラムを実行すると、while文の処理が永久に繰り返され、プログラムが終了しなくなってしまいます。条件の記述にあたっては十分注意してください。

重要

while文を使うと、繰り返し処理が記述できる。
繰り返し文の条件の記述に気をつける。

条件の記述を省略する

さてここで、if文やwhile文などで慣用的に使われている条件の書きかたについてみておくことにしましょう。次のコードをみてください。これはいったいどのようなプログラムなのでしょうか？

Sample6.cpp ▶ 慣用的な条件を使う

```
#include <iostream>
using namespace std;

int main()
{
    int num = 1;

    while(num){
        cout << "整数を入力してください。(0で終了) ¥n";
        cin >> num;
        cout << num << "が入力されました。¥n";
    }

    cout << "繰り返しが終わりました。¥n";

    return 0;
}
```

numが0であるとき（偽であるとき）繰り返しが終了します

Sample6の実行画面

```
整数を入力してください。(0で終了)
1 ↵
1が入力されました。
整数を入力してください。(0で終了)
10 ↵
10が入力されました。
整数を入力してください。(0で終了)
5 ↵
5が入力されました。
整数を入力してください。(0で終了)
0 ↵
0が入力されました。
繰り返しが終わりました。
```

このコードは、入力した整数を繰り返し出力するものです。この繰り返しは、ユーザーが0を入力したときに終了します。

さて、このコードのwhile文の条件をみてください。このwhile文は、

```
while(num){
...
```
numが0であるときにwhile文が終了します

となっていますね。これはどういう意味なのでしょうか？

第5章で説明したように、整数型の値はtrueまたはfalseに変換されます。したがって、ここでは、int型の変数numの値が0以外のときはtrue、0のときはfalseに変換されることになります。

つまりこのwhile文は「numが0であるとき」、すなわちユーザーが0を入力した場合にfalseとなり、終了することを意味しています。0以外の値の場合は、while文の処理を繰り返すわけです。

この条件を関係演算子を使って正しく書くと、次のように記述できます。


```
while(num != 0){
...
```
numが0であるときにwhile文が終了します

こちらは、きちんと関係演算子を使って入力された数値が0でないかどうかを判断しているわけです。

また、逆に、式が0以外の値のときに繰り返しを終了させる条件もよく使います。この条件は、次のように書きます。

```
while(!num){
...
```
numが0以外の値であるときにwhile文が終了します

!は否定をあらわす論理演算子です。この条件は、下のような条件と同じことをあらわしています。

```
while(num == 0){
...
```
numが0以外の値であるときにwhile文が終了します

6.3 do ～ while 文

do ～ while 文のしくみを知る

もうひとつ、繰り返しを行う構文をみてみましょう。ここでは、**do ～ while 文**（do-while statement）をとりあげます。この構文では、最後に指定した条件がtrueである間、ブロック内の処理を繰り返します。

構文　do~while 文

```
do{
    文1
    ...
}while(条件);
```

ブロック内の処理を順に繰り返し実行します

条件がtrueであれば繰り返しを続けます

do ～ while 文がwhile 文と異なるところは、

条件を判断する前にブロック内の処理を行う

ということです。while 文では、繰り返し処理の最初に条件がfalseであれば、一度もブロック内の処理が行われません。一方、do~while 文では、最低1回は必ずブロック内の処理が行われます。

たとえば、while 文でとりあげた例をdo ～ while 文に書きかえてみましょう。

```
do{
        試験を受ける
}while(試験に合格していない);
```

これは、while 文と同じように、試験を受け続ける繰り返し文です。しかしこちらは、処理をはじめる前に試験にすでに合格していても、最低一度は試験を受ける処理が行われるというわけです。while 文とみくらべてください。

図6-3 **do～while文**
while文ではブロック内の処理の前に条件が判断されますが、do～while文ではブロック内の処理のあとに条件が判断されます。

次のコードでは、Sample5.cppをdo~while文に書きかえてみました。

Sample7.cpp ▶ do～while文を使う

```
#include <iostream>
using namespace std;

int main()
{
   int i = 1;

   do{
      cout << i << "番目の繰り返しです。¥n";
      i++;
   }while(i <= 5);

   cout << "繰り返しが終わりました。¥n";

   return 0;
}
```

この部分が繰り返されます

i<=5がfalseであれば繰り返しを終了します

Sample7の実行画面

```
1番目の繰り返しです。
2番目の繰り返しです。
3番目の繰り返しです。
4番目の繰り返しです。
5番目の繰り返しです。
繰り返しが終わりました。
```

こちらは、do～while文を使っていますが、Sample5と同じ処理になっています。このように、同じ処理を行っていても、さまざまな構文を使って書きあらわせる場合があります。いろいろなスタイルのコードを記述する練習をしておくとよいでしょう。

重要

do～while文を使うと、繰り返し処理が記述できる。
do～while文は、最低1回はループ本体を実行する。

6.4 文のネスト

for文をネストする

これまでのところでいろいろな構文を学んできました。これらの条件判断文、繰り返し文などの構文では、複数の文を埋め込んでネストする（入れ子にする）ことができます。たとえば、次のように、for文の中にfor文を使うという複雑な記述ができるのです。

構文 **for文のネスト**

```
for(式1-1; 式2-1; 式3-1){
   ...
   for(式1-2; 式2-2; 式3-2){
      ...
   }
}
```

for文をネストすることができます

```
for(   ){

   for(   ){


   }
}
```

図6-4 **文のネスト**
for文などの構文はネストして記述することができます。

さっそく、for文をネストしたコードの例をみてみましょう。

Sample8.cpp ▶ for文をネストする

```
#include <iostream>
using namespace std;

int main()
{
   for(int i=0; i<5; i++){
      for(int j=0; j<3; j++){
         cout << "iは" << i << ":jは" << j << '¥n';
      }
   }

   return 0;
}
```

for文がネストされています

Sample8の実行画面

```
iは0:jは0
iは0:jは1
iは0:jは2
iは1:jは0
iは1:jは1
iは1:jは2
iは2:jは0
iは2:jは1
iは2:jは2
iは3:jは0
iは3:jは1
iは3:jは2
iは4:jは0
iは4:jは1
iは4:jは2
```

外側のループを1回処理するたびに、内側のループは3回処理されています

外側のループは全部で5回処理されています

このコードでは、変数iをインクリメントするfor文の中に、変数jをインクリメントするfor文を埋め込んで、入れ子にしています。このため、ループの中では次のような処理が行われています。

```
┌変数iをインクリメントする
│↓                変数jをインクリメントする ┐
│↓                変数jをインクリメントする │
└↓                変数jをインクリメントする ┘
┌変数iをインクリメントする
│↓                変数jをインクリメントする ┐
│↓                変数jをインクリメントする │
└↓                変数jをインクリメントする ┘
 ...
```

つまり、iをインクリメントするループ文が1回処理されるたびに、jをインクリメントするループ文の繰り返し（3回分）が行われるのです。このように、文をネストすると、複雑な処理でも記述できるようになります。

重要　for文をネストすると、多重の繰り返し処理が記述できる。

if文などと組みあわせる

上では、for文の中にfor文を埋め込みましたが、異なる種類の文を組みあわせてもかまいません。たとえば、for文とif文を組みあわせるようなこともできます。

次のプログラムを作成してください。

Sample9.cpp ▶ if文などと組みあわせる

```
#include <iostream>
using namespace std;

int main()
{
   int ch = 0;
```

```
    for(int i=0; i<5; i++){
        for(int j=0; j<5; j++){
            if(ch == 0){
                cout << '*';
                ch = 1;
            }
            else{
                cout << '-';
                ch = 0;
            }
        }
        cout << '¥n';
    }

    return 0;
}
```

for文がネストされています

*を出力したら、次は-を出力するように、chを1にします

-を出力したら、次は*を出力するように、chを0にします

内側のループが終わったら改行します

Sample9の実行画面

```
*-*-*
-*-*-
*-*-*
-*-*-
*-*-*
```

このコードでは、2つのfor文と1つのif～else文を使っています。*または-を出力するたびに、変数chに交互に0と1を代入します。こうすることによって、次にどちらの文字を出力するのかをif文中の「ch == 0」という条件を評価することで判断できます。

内側のループが終わると、¥nで改行していますので、5文字ごとに改行されています。文字の種類をかえてみたり、種類を増やすコードを考えてみたりしてください。

6.5 処理の流れの変更

break文のしくみを知る

これまでに学んだことから、繰り返し文には一定の処理の流れがあることがわかりました。しかし、ときには、このような処理の流れを強制的に変更したい場合があるかもしれません。

C++では、処理の流れを変更する文をとして、break文とcontinue文があります。この節では、まずはじめにbreak文を学ぶことにします。

break文（break statement）は、

処理の流れを強制的に終了し、そのブロックから抜ける

という処理を行う文です。次のようにコードの中に記述します。

構文 **break文**

```
break;
```

次のコードでは、break文を使って、キーボードから指定した回数で、繰り返しの処理を強制的に終了させてみます。

Sample10.cpp ▶ break文でブロックから抜ける

```
#include <iostream>
using namespace std;

int main()
{
   int res;
```

```
    cout << "何番目でループを中止しますか？（1～10）¥n";

    cin >> res;

    for(int i=1; i<=10; i++){
        cout << i << "番目の処理です。¥n";
        if(i == res)
            break;
    }

    return 0;
}
```

本来10回の繰り返しを行うfor文ですが・・・

指定した回数目で繰り返しを終了します

Sample10の実行画面

```
何番目でループを中止しますか？(1～10)
5 ↵
1番目の処理です。
2番目の処理です。
3番目の処理です。
4番目の処理です。
5番目の処理です。
```

指定回数で処理が終わります

Sample10では、本来全部で10回の繰り返しを行うfor文を使っています。しかしここでは、ユーザーが入力した回数でbreak文を実行し、ループを強制的に終了させてみました。6回目以降の繰り返し処理は行われていません。

なお、繰り返し文をネストしている場合、その内側の文でbreak文を使うと、内側のブロックを抜け出して、もうひとつ外側のブロックに処理がうつることになっています。

重要

break文を使って、ブロックから抜けることができる。

```
for(int i=1; i<=10; i++){
   cout << i << "番目の処理です。¥n";
   if(i == res)
      break;
}
```

図6-5 break文
break文を使うと、繰り返し処理を強制的に終了させてブロックから抜けることができます。

switch文の中でbreak文を使う

ところで、swtich文のところでみたように、switch文の中ではbreak文が使われていました。このとき使われるbreak文は、この節で説明しているbreak文と同じものです。つまり、switch文の中でbreak文を応用すると、次のような処理ができます。

Sample11.cpp ▶ switch文の中でbreak文を使う

```
#include <iostream>
using namespace std;

int main()
{
   int res;

   cout << "成績を入力してください。(1～5) ¥n";

   cin >> res;

   switch(res){
      case 1:
      case 2:
         cout << "もう少しがんばりましょう。¥n";
         break;
      case 3:
      case 4:
         cout << "この調子でがんばりましょう。¥n";
         break;
```

resが1や2のときはこの文が処理されます

break文の挿入位置に気をつけてください

resが3や4のときはこの文が処理されます

```
        case 5:
            cout << "たいへん優秀です。¥n";
            break;
        default:
            cout << "1～5までの成績を入力してください。¥n";
            break;
    }

    return 0;
}
```

Sample11の実行画面その1

```
成績を入力してください。(1～5)
1 ↵
もう少しがんばりましょう。
```

Sample11の実行画面その2

```
成績を入力してください。(1～5)
2 ↵
もう少しがんばりましょう。
```

Sample11の実行画面その3

```
成績を入力してください。(1～5)
3 ↵
この調子でがんばりましょう。
```

Sample11は、入力した整数の成績によって、いろいろなメッセージを表示するプログラムです。コード中のbreak文の挿入位置に注意してください。case 1とcase 3にはbreak文がありませんので、それぞれcase 2あるいはcase 4と同じ処理をするようになっています。このように、break文の挿入位置によって処理をコントロールできるのです。

continue文のしくみを知る

もうひとつ、文の流れを強制的に変更する文として、**continue文**(continue statement)をみておきましょう。continue文は、

ブロック内の処理を飛ばし、ブロックの先頭位置に戻って次の処理を続ける

という文です。

構文 **continue文**

```
continue;
```


continue文を使ったコードをみてみましょう。

Sample12.cpp ▶ continue文でブロックの最初に戻る

```
#include <iostream>
using namespace std;

int main()
{
   int res;

   cout << "何番目の処理を飛ばしますか?(1～10)¥n";

   cin >> res;

   for(int i=1; i<=10; i++){
      if(i == res)
         continue;
      cout << i << "番目の処理です。¥n";
   }

   return 0;
}
```

入力した回数目の処理では、ここから先頭に戻ります

入力した回数目では、この文は処理されません

Sample12の実行画面

```
何番目の処理を飛ばしますか？(1～10)
3 ↵
1番目の処理です。
2番目の処理です。
4番目の処理です。
5番目の処理です。
6番目の処理です。
7番目の処理です。
8番目の処理です。
9番目の処理です。
10番目の処理です。
```

3番目の繰り返し処理ではcontinue文のあとが飛ばされているので、出力されていません

Sample12を実行して、処理を飛ばす回数として「3」を入力してみました。すると、3番目の繰り返し処理は、continue文の処理が行われることによって強制的に終了させられ、ブロックの先頭、つまり次の繰り返し処理にうつります。したがって、上では「3番目の処理です。」という出力は行われていません。

重要 continue文を使って、次の繰り返し処理にうつることができる。

```
for(int i=1; i<=10; i++){

   if(i == res)
      continue;
   cout << i << "番目の処理です。¥n";

}
```

図6-6 continue文
繰り返し内の処理を飛ばし、次の繰り返しにうつるには、continue文を使います。

goto文

このほかに、処理の流れを強制的に変更する構文として、**goto文**（goto statement）があります。goto文は、識別子を記述した場所に処理の流れを変更します。

構文 goto文

```
goto 識別子;

識別子:
    文;
```

この部分にジャンプします

たとえば、プログラムの中では次のような処理になります。

```
...
goto ABC;
...

ABC:
...
```

ただし、goto文は適当な場所にプログラムの流れを変更するものですから、コードがたいへん読みにくくなります。できるだけgoto文を使わないでコードを記述するほうがよいでしょう。

6.6 レッスンのまとめ

この章では、次のようなことを学びました。

- for 文を使うと、繰り返し処理ができます。
- while 文を使うと、繰り返し処理ができます。
- do ~ while 文を使うと、繰り返し処理ができます。
- 文はネストすることができます。
- break 文を使うと、ブロック内の処理を終了します。
- continue 文を使うと、繰り返し文の最初に戻って次の繰り返し処理にうつります。

この章では繰り返し処理や、処理の流れを変更する構文を学びました。第5章で学んだ構文とあわせて使うと、さまざまな処理を行う複雑なプログラムを記述することができます。プログラムで行おうとする処理を、これらの文にあてはめて自由自在に記述できるように練習してみてください。

練習

1. 次のように画面に出力するコードを記述してください。

```
1～10までの偶数を出力します。
2
4
6
8
10
```


2. キーボードからテストの点数を入力させ、その合計点を出力するコードを記述してください。最後に答えを出力させる場合には、0を入力するものとします。

```
テストの点数を入力してください。(0で終了)
52 ↵
68 ↵
75 ↵
83 ↵
36 ↵
0 ↵
テストの合計点は314点です。
```

3. 次のように画面に出力するコードを記述してください。

```
*
**
***
****
*****
```

Lesson 7

関数

これまでのところで、C++のさまざまな機能を学び、複雑なコードを記述できるようになってきました。プログラムが大きくなってくると、コード中の複数の場所で同じような処理を行わなければならない場合があります。大きなプログラムを作成する場合には、一定の処理をまとめておき、あとでその処理を呼び出す機能が重要となってきます。この章では、複数の処理をまとめる「関数」という機能について学びましょう。

Check Point!

- 関数の定義
- 関数の呼び出し
- 引数
- 戻り値
- インライン関数
- 関数の宣言
- デフォルト引数
- 関数のオーバーロード
- 関数テンプレート

7.1 関数

関数のしくみを知る

私たちは日常生活の中で、一定のまとまった処理を何度も繰り返して行うことがあります。たとえば、毎月自分の貯金からお金を引き出す場合について考えてみてください。このとき、預金を引き出すたびに次のような処理を行っていますね。

通帳を自動現金支払機に入れる
暗証番号を入力する
金額を指定する
お金を受けとる
お金と通帳を確認する

C++でも、複雑なコードを書くようになってくると、たびたび行わなければならない一定の処理が出てくる場合があります。このような処理を、そのたびに何度も記述していくのはとても面倒です。

そこでC++は、

一定の処理をまとめて記述する

関数（function）という機能を用意しています。

関数を利用すると、複数の処理をひとまとめにして、いつでも呼び出して使うことができるようになります。たとえば、上にあげたような、預金をおろす一連の処理を「関数」としてまとめておくのです。この関数には、

引き出し処理

などという名前をつけることにします。そうすると、このまとめた関数は「引き出し処理」という1つの処理として、あとでかんたんに呼び出せるようになります。

図 7-1 関数を作成する
関数（アミ部分）を作成することによって、まとめた処理をかんたんに呼び出すことができるようになります。

C++で関数を利用するためには、次の2つの手順をふむことが必要です。

1. **関数を作成する（関数を定義する）**
2. **関数を利用する（関数を呼び出す）**

この章では、まず最初に「関数を定義する」作業からみていくことにしましょう。

main()関数

大きな目でみれば、プログラム全体も「ひとまとまりの処理」です。したがって、C++のプログラム全体も、1つの関数となっています。この関数の名前は、もうおなじみになったmain()という名前の関数です。

7.2 関数の定義と呼び出し

関数を定義する

関数を使うには、まず最初に、コードの中で一定のまとまった処理を指定しなければなりません。これが関数を作成する作業になります。この作業は、

関数の定義（function definition）

と呼ばれています。関数の定義は、ブロック内にまとめて記述します。次のコードが、関数の一般的なスタイルです。

構文　関数の定義

```
戻り値の型 関数名（引数リスト）
{
   文;
   ...

 return 式;
}
```

関数の名前には識別子を使います

処理をまとめておきます

「戻り値」や「引数」という耳慣れない用語が使われていますが、これらの用語についてはあとでくわしく説明することにしましょう。ここでは関数の大まかなイメージだけをながめておいてください。

また、「関数名」とは、変数の名前と同じように、識別子（第3章）を使ってつけた関数の名前です。

たとえば、次のようなコードが関数の定義です。これは画面に「車を買いました。」と出力する処理を行う「buy」という名前の関数です。

```
//buy関数の定義
void buy()
{
    cout << "車を買いました。¥n";
}
```

関数の名前です

処理をブロック内にまとめました

buyという関数名をつけて、ブロックの中に1文からなる処理を記述していますね。なお、ブロックの最後の}のあとにはセミコロン（;）をつけたりしないので、注意してください。

重要

関数を定義して、一定の処理をまとめておくことができる。


```
void buy()
{
    cout << "車を買いました。¥n";
}
```

関数の定義

図7-2 関数の定義
一定の処理をまとめて、関数を定義することができます。

関数を呼び出す

関数を定義すると、このまとめた処理をあとで利用することができるようになります。関数を利用することを、

関数の呼び出し（function call）

とも呼んでいます。

では、関数を呼び出す方法を学びましょう。関数を呼び出すには、コード中でその関数名を次のように記述します。

構文 **関数の呼び出し**

```
関数名(引数リスト);
```

たとえば、前ページで定義した関数を呼び出すには、

```
buy();
```

と記述するのです。今度は、ブロックを記述しません。最後にセミコロンを忘れずにつけてください。コードの中でこの関数の呼び出しが処理されると、さきほど定義した関数の処理がまとめて行われることになっています。

それでは次のコードを入力して、関数の定義と呼び出し方法をたしかめてみましょう。

Sample1.cpp ▶ 基本の関数を作成する

```
#include <iostream>
using namespace std;

//buy関数の定義
void buy()
{
    cout << "車を買いました。¥n";
}

//buy関数の利用
int main()
{
    buy();

    return 0;
}
```

buy()関数の処理内容です

ここでbuy()関数の処理が行われます

Sample1 の実行画面

```
車を買いました。
```

buy()関数の処理が行われています

Sample1のコードは、次の2つの部分に分かれていることに注目してください。

- main()関数の部分
- buy()関数の部分

C++ではmain()関数から処理がはじまるのでしたね。そこで、ここでもこれまでどおり、プログラムが実行されると、main()関数の先頭部分から処理が行われます。

main()関数の最初の文では、buy()関数を呼び出しています（①）。すると処理の流れはbuy()関数にうつって、関数の最初の文から順番に処理が行われていきます（②）。その結果、画面には「車を買いました。」という文字列が出力されることになります。

buy()関数内の処理は、}がくると終わりです。buy()関数が終わると、さきほどのmain()関数の処理に戻ることになっています（③）。ここがmain()関数の最後の部分ですから、これでプログラムが終了します。

つまり、関数を利用するコードでは、

① 関数を呼び出す

↓

② 関数内部の処理が行われる

↓

③ 関数ブロック内の処理が終わると、呼び出し元に処理が戻る

という流れで処理が行われるのです。Sample1の処理の流れをまとめてみると、図7-3のようになっています。

重要

関数を呼び出すと、定義しておいた処理がまとめて行われる。

```
void buy()
{
    cout << "車を買いました。¥n";
}

int main()
{
    buy();

    return 0;
}
```


図7-3 関数の呼び出し
①関数を呼び出すと、②関数内部の処理が行われます。③内部の処理が終わると呼び出し元に戻って処理を続けます。

関数を何度も呼び出す

関数の流れをさらにつかむために、もうひとつ、次のようなコードを作成してみましょう。今度は関数を2回呼び出してみることにします。

Sample2.cpp ▶ 関数を何度も呼び出す

```
#include <iostream>
using namespace std;

//buy関数の定義
void buy()
{
    cout << "車を買いました。¥n";
}

//buy関数の利用
int main()
{
    buy();

    cout << "もう1台車を購入します。¥n";

    buy();

    return 0;
```

```
}
```

Sample2の実行画面

```
車を買いました。
もう1台車を購入します。
車を買いました。
```

関数が2回呼び出されています

このコードでは、まず、main()関数の1文目で、buy()関数の処理が行われます(①)。この処理が終ったらmain()関数に戻るので、「もう1台車を購入します。」という文字列が出力されます(②～④)。

そして、もう一度、buy()関数の呼び出しが行われます(⑤)。今度も同じことの繰り返しが行われます(⑥～⑦)。

実行結果をみると、2回関数が呼び出されているようすがわかります。関数の処理の流れを追いかけてみてください。

重要

関数は何度でも呼び出すことができる。

```
void buy()
{
   cout << "車を買いました。¥n";
}

int main()
{
   buy();

   cout << "もう1台車を購入します。¥n";

   buy();

   return 0;
}
```

①②③④⑤⑥⑦

図7-4　複数回の呼び出し
関数は複数回呼び出すことができます。

関数に処理をまとめる

ところで、このような関数の処理を、ちょっと面倒に思われる方もいるかもしれません。これまで見てきたbuy()関数は、たった1つの文を処理するだけなので、関数としてわざわざ定義する必要性はあまりありません。関数を使わないで、main()関数内に画面出力の処理をすべて記述したほうがかんたんです。つまり、次のようなコードを記述してもSample2と同じ結果となります。

```
#include <iostream>
using namespace std;

int main()
{
    cout << "車を買いました。¥n";
    cout << "もう1台車を購入します。¥n";
    cout << "車を買いました。¥n";

    return 0;
}
```

Sample2と同じ処理といえます

しかし、もっと複雑なコードではどうでしょうか。関数には、さまざまな複雑な処理を定義することができます。たとえば、次のようなbuy()関数を考えてみてください。

```
//buy関数の定義
void buy()
{
    cout << "通帳を自動受払い機にいれます。¥n";
    cout << "暗証番号を入力します。¥n";
    cout << "金額を指定します。¥n";
    cout << "お金を受けとります。¥n";
    cout << "お金と通帳を確認します。¥n";
    cout << "車を買いました。¥n";
}
```

複雑な処理もまとめられます

このbuy()関数は、6つの文からなる処理をまとめたものです。このような複雑

な処理も、まとめて定義しておけば便利です。車を購入する処理を行いたいときには、次のように短い関数名を使って、一連の処理をかんたんに呼び出すことができるからです。

```
buy();
```

この短い呼び出しだけで複雑な処理が行えます

車を購入するたびに、6つの文を何度も記述する必要はありません。関数を使えば、たった1つの文を記述するだけでよいのです。

また、main()関数にさまざまな処理をえんえんと記述していくと、大きなプログラムの中では、どのような処理が行われているのかが、非常にわかりにくくなります。

関数を使えば、まとまった処理に名前をつけて記述することで、わかりやすいコードとすることができます。関数は複雑なプログラムを記述するために、欠かせない機能なのです。

重要

関数を使って、複雑なプログラムをわかりやすく作成することができる。

7.3 引数

引数を使って情報を渡す

この節では、関数について、さらにくわしくみていくことにしましょう。関数は処理をまとめることに加えて、さらに柔軟な処理を行う方法も用意されています。

関数を呼び出す際に、

呼び出し元から関数内に何か情報（値）を渡し、
その値に応じた処理を行う

という処理ができるようになっているのです。関数に渡す情報を**引数**（argument）といいます。引数を使う関数は、次のようなかたちで記述します。

```
//buy関数の定義
void buy(int x)
{
   cout << x << "万円の車を買いました。¥n";
}
```

int型の引数を用意します

引数を関数内で使います

このbuy()関数は、呼び出し元から呼び出されるときに、int型の値を1つ、関数内に渡すように定義したものです。関数の()内にある「int x」が、引数と呼ばれるものです。引数xは、この関数内だけで使うことができるint型の変数となっています。

変数x（引数）は、関数が呼び出されたときに、呼び出し元から渡されるint型の値が格納されます。このため、変数xの値は関数内部の処理に利用することができるようになっています。buy()関数では、渡された値を出力する、という処理を行っているわけです。

```
void buy(int x)
{
   cout << x << "万円の車を買いました。¥n";
}
```

図7-5 引数
関数の本体に情報（引数）を渡して処理することができます。

なお、変数xは、buy()関数内以外の場所では使うことができません。buy()関数以外の場所で、値を代入したり、出力したりすることができないのです。つまり、この変数xをmain()関数の中では使うことはできないので注意してください。

引数を使って、関数に値を渡すことができる。

引数を渡して関数を呼び出す

では実際に、引数をもつbuy()関数を呼び出してみましょう。引数をもつ関数を呼び出すときには、関数呼び出し文の()の中に、指定しておいた型の値を記述して、関数に値を渡します。

Sample3.cpp ▶ 引数をもつ関数を使う

```
#include <iostream>
using namespace std;

//buy関数の定義
void buy(int x)
{
```

値を受けとる仮引数です

```
    cout << x << "万円の車を買いました。¥";
}

//buy関数の呼び出し
int main()
{
    buy(20);
    buy(50);

    return 0;
}
```

受けとった値を出力します

実引数として20を渡して呼び出します

実引数として50を渡して呼び出します

Sample3の実行画面

```
20万円の車を買いました。
50万円の車を買いました。
```

渡した値が出力されます

このmain()関数内では、

最初にbuy()関数を呼び出すとき、「20」という値を渡して呼び出す
次にbuy()関数を呼び出すとき、「50」という値を渡して呼び出す

という処理を行っています。値はbuy()関数の引数xに渡されて格納されます。20を渡したときは20を、50を渡したときは50が出力されています。関数を呼び出すたびに、渡した引数に応じた金額が出力されているのがわかりますね。

このように同じ関数でも、渡された引数の値によって異なる処理を行うことができます。引数を使えば、柔軟な関数を作成できるのです。

なお、関数の本体で定義されている引数（変数）を**仮引数**（parameter）と呼びます。一方、関数の呼び出し元から渡される引数（値）を、**実引数**（argument）と呼びます。ここでは変数xが仮引数、「20」や「50」が実引数というわけです。

重要

関数の定義内で値を受けとる変数を、仮引数という。
関数を呼び出す際に渡す値を、実引数という。

20

x

仮引数

```
void buy(int x)
{
   cout << x << "万円の車を買いました。¥n";
}
```

20

```
int main()
{

   buy(20);

}
```

実引数

20

図 7-6 仮引数と実引数

関数には仮引数を定義しておくことができます。関数呼び出しの際に
実引数を渡して処理することができます。

キーボードから入力する

引数についてさらに理解するために、次のようにキーボードから入力した値を関数内に渡してみることにしましょう。次のコードを作成してみてください。

Sample4.cpp ▶ 実引数を変数の値とする

```
#include <iostream>
using namespace std;

//buy関数の定義
void buy(int x)
{
   cout << x << "万円の車を買いました。¥";
}
```

```
//buy関数の呼び出し
int main()
{
    int num;

    cout << "1台目はいくらの車を買いますか? ¥";
    cin >> num;

    buy(num);    ← 引数として変数num(の値)を渡します

    cout << "2台目はいくらの車を買いますか? ¥";
    cin >> num;

    buy(num);    ← もう一度変数num(の値)を渡します

    return 0;
}
```

Sample4の実行画面

```
1台目はいくらの車を買いますか?
20 ↵
20万円の車を買いました。        ← 渡された値が出力されます
2台目はいくらの車を買いますか?
50 ↵
50万円の車を買いました。
```

ここでは呼び出し元から関数に渡す実引数として、main()関数内で用意した変数num(の値)を使いました。キーボードからnumに読み込んだ値を関数に渡すようにしたわけです。

このように、実引数として変数を使うときには、実引数と仮引数の変数名は同じでなくてもかまいません。ここでは異なる変数名を使ってコードを記述しています。

なお、C++では、関数に渡されるのは実引数そのものではなく、そこに格納されている「値」となっています。このような引数の渡しかたを**値渡し**(pass by value)と呼ぶことがあります。

重要

関数が呼び出されると、実引数の「値」が渡される。


```
void buy(int x)
{
   cout << x << "万円の車を買いました。¥n";
}
int main()
{

   buy(num);


}
```

図7-7 値渡し
関数が呼び出されると、実引数（の値）が渡されて仮引数の初期化が行われます。

複数の引数をもつ関数を使う

さて、これまでに定義した関数の引数は1個だけでした。関数には、さらに2個以上の引数をもたせることもできます。関数を呼び出すときに、複数の値を関数内に渡して処理することができるのです。さっそくコードを作成してみることにしましょう。

Sample5.cpp ▶ 複数の引数をもつ関数を使う

```
#include <iostream>
using namespace std;

//buy関数の定義
void buy(int x, int y)
{
   cout << x << "万円と" << y << "万円の車を買いました。¥n";
}

//buy関数の呼び出し
int main()
{
   int num1, num2;

   cout << "いくらの車を買いますか？ ¥n";
   cin >> num1;

   cout << "いくらの車を買いますか？ ¥n";
   cin >> num2;

   buy(num1, num2);

   return 0;
}
```


Sample5の実行画面

```
いくらの車を買いますか？
20 ↵
いくらの車を買いますか？
50 ↵
20万円と50万円の車を買いました。
```


複数の引数をもつ関数も、これまでと基本は同じです。ただし、呼び出す際に複数の引数をカンマ（,）で区切って指定してください。この複数の引数を**引数リスト**と呼ぶこともあります。すると、区切った順に、実引数の値が仮引数に渡されます。つまり、Sample5のbuy()関数では、次のように値が渡されるのです。

仮引数　　　実引数
　x　　←　　num1の値
　y　　←　　num2の値

関数内では受けとった2つの値を出力する処理をしているのがわかりますね。

重要

関数には複数の引数を渡すことができる。

```
void buy(int x, int y)
{
   cout << x << "万円と" << y << "万円の車を買いました。¥n";
}

int main()
{
   buy(num1, num2);
}
```


図7-8 **複数の引数**
引数は複数指定することができます。引数リストの順に値が渡されます。

なお、仮引数と異なる数の実引数を渡して関数を呼び出すことはできません。たとえば、2つの引数を使うbuy()関数を定義した場合は、引数を1つだけ指定して呼び出すことはできないので、注意してください。

```
//buy関数の呼び出し
...
buy(num1, num2);   ← 仮引数の数にみあった実引数を渡します
buy(num1);         ← この呼び出しは誤りです
```

引数のない関数を使う

ところで、関数の中には、この章の最初で定義したbuy()関数のように、「引数のない関数」というものもあります。引数のない関数を定義するときには、無指定とするか、または引数の型としてvoidという特殊な型名を指定します。

```
//buy関数の定義
void buy()        ← 引数をもたない場合は無指定またはvoid型とします
{
   cout << "車を買いました。¥n";
}
```

このような関数を呼び出す際には、()内に値を指定しないで呼び出します。この章の最初では、引数を指定しないでbuy()関数を呼び出していたことを復習してみてください。これが引数のない関数の呼び出しかたになります。

```
//buy関数の呼び出し
...
buy();   ← 引数を渡さないで呼び出します
```

重要

引数のない関数では、無指定またはvoid型とする。

```
void buy()
{
   cout << "車を買いました。¥n";
}

int main()
{

   buy();


}
```

図7-9 引数のない関数

引数のない関数を作成することもできます。引数はvoid型または無指定としておきます。

7.4 戻り値

戻り値のしくみを知る

引数の使いかたが理解できたでしょうか。

関数では、引数とちょうど逆に、

関数の呼び出し元に、関数本体から特定の情報を返す

というしくみも、作成することができるようになっています。

関数から返される情報を、**戻り値**（return value）といいます。複数指定できる引数と違って、戻り値はただ1つだけ、呼び出し元に値を返すことができます。

8.2節で紹介した関数の定義のスタイルをもう一度みてください。戻り値を返すには、まず戻す値の「型」を関数の定義の中に示しておきます（①）。そして関数のブロックの中で、returnという文を使って、実際に値を返す処理をします（②）。

構文　関数の定義

```
戻り値の型 関数名(引数リスト)     ① 戻り値の型を指定します
{
    文;
    ...

    return 式;                    ② 式の値を呼び出し元に返します
}
```

ここではブロックの最後にreturn文を記述していますが、この文はブロックの中ほどに記述することもできます。ただし、関数が処理されたとき、ブロックの最後まで行きつかなくても、return文が処理されたところでその関数の処理が終了します。return文のしくみに気をつけておくことが必要です。

```
int buy(int x, int y)
{
   ...
   return z;
}
```


図7-10 戻り値
関数内から戻り値を呼び出し元に返すことができます。

では、戻り値をもつ関数をみてみましょう。次のコードが戻り値をもつ関数です。

```
//buy関数の定義
int buy(int x, int y)
{
   int z;

   cout << x << "万円と" << y << "万円の車を買いました。¥n";

   z = x+y;

   return z;
}
```


この関数では、受けとった2つの引数xとyをたしあわせる処理をしています。たし算の結果は、関数内で宣言した変数zに格納します。

そしてreturn文で、zの値を戻り値として返す処理をします。zはint型ですので、戻り値の型にはint型と指定してあります。

実際にコードを記述して、この関数を使ってみることにしましょう。

Sample6.cpp ▶ 戻り値をもつ関数

```
#include <iostream>
using namespace std;

//buy関数の定義
int buy(int x, int y)
{
```

```
    int z;

    cout << x << "万円と" << y << "万円の車を買いました。¥n";

    z = x+y;

    return z; ●──── 戻り値を返します
}

//buy関数の呼び出し
int main()
{
    int num1, num2, sum;

    cout << "いくらの車を買いますか？ ¥n";
    cin >> num1;

    cout << "いくらの車を買いますか？ ¥n";
    cin >> num2;

    sum = buy(num1, num2); ●──── 関数を呼び出し、その戻り値を変数sumに代入します

    cout << "合計で" << sum << "万円です。¥n";
                         └──── 戻り値の値を出力します
    return 0;
}
```

Sample6の実行画面

```
いくらの車を買いますか？
20 ↵
いくらの車を買いますか？
50 ↵
20万円と50万円の車を買いました。
合計で70万円です。 ●──── 戻り値が出力されています
```

ここでは、関数内で計算された結果の戻り値を、呼び出し元のsumという変数に格納するようにしています。戻り値を利用するには、関数の呼び出し文から、代入演算子を使って代入してください。

```
//buy関数の呼び出し
...
sum = buy(num1, num2);
```
戻り値を変数sumに代入します

呼び出し元ではこの変数sumの内容を出力しています。このように、関数の戻り値を変数に代入して、呼び出し元で利用することができるのです。

なお、戻り値は必ずしも呼び出し元で利用しなくてもかまいません。戻り値を利用しないときには、

```
buy(num1, num2);
```
戻り値は利用しなくてもかまいません

とだけ記述します。

重要 戻り値を使うと、関数の呼び出し元に情報を返すことができる。

```
int buy(int x, int y)
{
    ...
    return z;
}
int main()
{

    sum = buy(num1, num2);

}
```


図7-11 戻り値の利用
呼び出し元では戻り値を使って処理を行うことができます。

戻り値のない関数

引数のない関数を定義することができたように、戻り値のない関数も定義することができます。たとえば、7.2節で定義したbuy()関数は、戻り値をもたない関数です。

```
//buy関数の定義
void buy()
{
   cout << "車を買いました。¥n";
}
```

戻り値をもたない場合はvoid型とします

関数が戻り値をもたないことを示すためには、戻り値の型として**void型**を指定しておきます。戻り値のない関数が呼び出されて処理された場合には、ブロックの最後の } まで行きつくか、次のような何もつけないreturn文によって関数が終了することになっています。

構文 **return文**

```
return;
```

上のbuy()関数をreturn文を使って記述してみました。ただし、このような単純な関数ではreturn文は記述してもしなくても同じです。

```
//buy関数の定義
void buy()
{
   cout << "車を買いました。¥n";

   return;
}
```

呼び出し元の処理に戻ります

```
void buy()
{
   cout << "車を買いました。¥n";
}
```

```
int main()
{

   buy();

}
```

図7-12 戻り値のない関数

戻り値をもたない関数では、関数の戻り値の型にvoidを指定しておきます。

7.5 関数の利用

合計値を求める関数

関数のかたちと処理の流れが理解できたでしょうか？ この節では、これまでの知識を利用して、さまざまな処理を行う関数を作成してみることにしましょう。まず最初に、前の節で記述したbuy()関数を、より広く使える関数にしてみましょう。2つの数の合計値を求める関数にしてみます。

Sample7.cpp ▶ 合計値を求める関数

```
#include <iostrem>
using namespace std;

//sum関数の定義
int sum(int x, int y)
{
   return x+y;
}

int main()
{
   int num1, num2, ans;

   cout << "1番目の整数を入力してください。¥n";
   cin >> num1;

   cout << "2番目の整数を入力してください。¥n";
   cin >> num2;

   ans = sum(num1, num2);

   cout << "合計は" << ans << "です。¥n";

   return 0;
}
```

Sample7の実行画面

```
1番目の整数を入力してください。
10 ↵
2番目の整数を入力してください。
5 ↵
合計は15です。
```

合計値が出力されています

ここでは、2つの数値の合計値を求めるsum()関数を定義しました。この関数の処理は、前の節で説明した戻り値をもつbuy()関数と、実際には同じです。前の節ではわかりやすいように合計値をいったん変数に格納してから、その値を返すようにしましたが、こちらでは合計を求める式を使ってそのまま値を返しています。よりかんたんにコードを記述しているわけです。

```
return x+y;
```

合計値を戻り値として返しています

最大値を求める関数

ではもうひとつ、便利な関数を定義する練習をしてみましょう。次のようなコードを入力してみてください。

Sample8.cpp ▶ 最大値を求める関数

```
#include <iostream>
using namespace std;

//max関数の定義
int max(int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
}

int main()
```

2つの数を受けとります
xがyより大きい場合には・・・
xの値を返します
そうでない場合は、yの値を返します

```
{
    int num1, num2, ans;

    cout << "1番目の整数を入力してください。¥n";
    cin >> num1;

    cout << "2番目の整数を入力してください。¥n";
    cin >> num2;

    ans = max(num1, num2);

    cout << "最大値は" << ans << "です。¥n";

    return 0;
}
```

関数を呼び出します

戻り値を出力します

Sample8 の実行画面

```
1番目の整数を入力してください。
10 ↵
2番目の整数を入力してください。
5 ↵
最大値は10です。
```

最大値が出力されています

再帰

なお、C++ 関数では、その内部で自分自身を呼び出す処理をすることができるようになっています。このようなしくみを**再帰**(recursion) と呼ぶ場合があります。

```
void func()
{
    ...
    func();
}
```

自分自身の処理を呼び出します

再帰を使うと、複雑な処理がかんたんに記述できる場合があります。

今度は2つの数のうち、大きいほうの値を返すmax()関数を定義しました。この関数は、変数xまたはyどちらかの値を呼び出し元に返しています。どちらかのreturn文が処理されると、その場所で関数の処理が終了して、呼び出し元の処理に戻るのです。

実行結果をみると、2つの値の最大値を出力する処理が行われていますね。このように、使いやすい関数を作成しておくことで、いろいろなコードで関数を利用することができるわけです。

インライン関数のしくみを知る

ところで、関数を使ったコードでは、関数を使わない場合に比べて処理を実行するのに余分な時間がかかる場合があります。処理を関数としてまとめると、引数や戻り値の受け渡しなどの際に、余分な時間がかかるのです。何度も呼び出されるような小さな関数の場合には、この時間が無視できない場合もあります。

そのようなときには、**インライン関数**（inline function）というものを利用すると便利です。インライン関数とは次のようなかたちをした関数です。

構文 **インライン関数の定義**

```
inline 戻り値の型 関数名(引数リスト) { … }
```

インライン関数は、先頭にinlineをつけた関数です。

実際に、インライン関数を使ったコードを入力してみることにしましょう。

Sample9.cpp ▶ インライン関数を使う

```
#include <iostream>
using namespace std;

//max関数の定義
inline int max(int x, int y){if(x>y) return x; else return y;}

int main()
{
```

インライン関数を定義します

Lesson 7

```
    int num1, num2, ans;

    cout << "1番目の整数を入力してください。¥n";
    cin >> num1;

    cout << "2番目の整数を入力してください。¥n";
    cin >> num2;

    ans = max(num1, num2);

    cout << "最大値は" << ans << "です。¥n";

    return 0;
}
```

インライン関数を呼び出した部分に、最大値を求めるコードが埋め込まれます

Sample9の実行画面

```
1番目の整数を入力してください。
10 ↵
2番目の整数を入力してください。
5 ↵
最大値は10です。
```

この実行結果はSample8のmax()関数を使ったコードと同じです。ただし、こちらの場合は、図7-13のように、インライン関数を呼び出した場所に最大値を求めるコードがコンパイラによって埋め込まれることになります。インライン関数の処理は呼び出し部分に埋め込まれるので、プログラムの処理速度が向上することがあります。

ただし、コンパイラは長い処理についてはインライン関数として扱わない場合もあります。コンパイラは短い処理だけをインライン関数として埋め込むしくみになっています。

重要

関数を、インライン関数とすることができる。

maxを求める
maxを求める
maxを求める

maxを求める

図7-13 **インライン関数**
インライン関数（左）では、通常の関数（右）と異なり、使用した部分にコードが埋め込まれます。

7.6 関数の宣言

関数を宣言する

さて、これまでに定義した関数は、その関数を呼び出すコードの「前の部分」に記述してきました。ではもし、関数の定義を呼び出しよりも前に記述すると、どうなるのでしょうか？ つまり、main()関数の前に記述するとどうなるのでしょうか？

実はこのようなコードを作成すると、

関数が定義されていません

という内容のエラーが出て、コードをコンパイルすることができません。

○

```
int max(int x, int y)
{
...
}
```

関数の定義

```
int main()
{
   int ans = max(num1, num2);
}
```

関数の呼び出し

×

```
int main()
{
   int ans = max(num1, num2);
}
```

関数の呼び出し

```
int max(int x, int y)
{
...
}
```

関数の定義

図7-14 関数を定義する場所
関数を呼び出す前に関数の定義をしないと、コンパイルできません。

C++では、関数の呼び出しを先に記述する場合には、

関数の名前と引数の数をコンパイラに知らせておく

という作業が必要です。これを**関数プロトタイプ宣言**（関数宣言：function declaration）と呼びます。つまり、関数の呼び出しを関数の定義よりも前に記述する場合には、関数プロトタイプ宣言が必要なのです。

関数プロトタイプ宣言は、関数を呼び出す前に、呼び出す関数の名前・戻り値の型・引数を次のように記述します。

構文 **関数プロトタイプ宣言**

```
戻り値の型 関数名(引数リスト);
```

それでは関数プロトタイプ宣言を使ったコードをみてみましょう。関数本体はmain()関数のあとに定義することにします。

Sample10.cpp ▶ 関数プロトタイプ宣言を使う

```
#include <iostream>
using namespace std;

//max関数の宣言
int max(int x, int y);          ← 関数プロトタイプ宣言です


//max関数の呼び出し
int main()
{
   int num1, num2, ans;

   cout << "1番目の整数を入力してください。¥n";
   cin >> num1;

   cout << "2番目の整数を入力してください。¥n";
   cin >> num2;

   ans = max(num1, num2);       ← 関数を呼び出しています

   cout << "最大値は" << ans << "です。¥n";
```

```
    return 0;
}

//max関数の定義
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

関数の定義をあとに書くことができます

Sample10の実行画面

```
1番目の整数を入力してください。
5 ↵
2番目の整数を入力してください。
10 ↵
最大値は10です。
```

Sample10では、max()関数が2つの引数をもち、int型の戻り値を返すことを、関数プロトタイプ宣言で示しました。もし、あとに記述したmax()関数の戻り値や引数がプロトタイプ宣言と異なる場合は、コンパイルするときにエラーが出るようになっています。このような関数プロトタイプ宣言のかたちをおぼえておいてください。

```
int max(int x, int y);
```

関数プロトタイプ宣言

○

```
int main()
{
    int ans = max(num1, num2);
}
```

関数を利用する（呼び出す）

```
int max(int x, int y)
{
        ...
}
```

関数の本体の定義

図7-15 関数プロトタイプ宣言
関数を呼び出す前に、関数の仕様を宣言しておく必要があります。

重要

関数の呼び出しよりもあとの部分に関数の定義を記述する場合には、関数を宣言する必要がある。

大規模なプログラムを作成する

ここでは、関数の本体をあとで記述するために関数プロトタイプ宣言を使いました。しかし、関数プロトタイプ宣言が最も威力を発揮するのは、大規模なプログラムを作成する場合です。この方法については第10章で説明します。

デフォルト引数を使う

関数プロトタイプ宣言では、**デフォルト引数**（default argument）を指定しておくことがよく行われます。デフォルト引数を指定しておくと、関数を呼び出す際に実引数を省略することができます。実引数を省略すると、指定しておいたデフォルト値が関数に渡されるのです。

デフォルト引数を指定した関数プロトタイプ宣言は、次のようになります。

構文 **デフォルト引数の指定**

```
戻り値の型名 関数名(型名 仮引数名=デフォルト値, ・・・ );
```

デフォルト引数は、関数の宣言または定義に1回だけ指定します。つまり、関数プロトタイプ宣言でデフォルト引数を指定した場合は、関数本体の定義ではデフォルト引数を指定できません。

次の関数プロトタイプ宣言は、引数に10というデフォルト引数を指定したものです。

```
void buy(int x=10);
```

デフォルト引数を指定します

それでは次のコードで、デフォルト引数をもつ関数を、実際に呼び出してみましょう。

Sample11.cpp ▶ デフォルト引数を使う

```
#include <iostream>
using namespace std;

//buy関数の宣言
void buy(int x=10);

//buy関数の呼び出し
int main()
{
    cout << "1回目は100万円で購入します。¥n";
    buy(100);

    cout << "2回目はデフォルト金額で購入します。¥n";
    buy();

    return 0;
}

//buy関数の定義
void buy(int x)
{
    cout << x << "万円の車を買いました。¥n";
}
```

buy(100); — 引数を指定して呼び出します

buy(); — 引数を指定しないで呼び出します

Sample11の実行画面

```
1回目は100万円で購入します。
100万円の車を買いました。
2回目はデフォルト金額で購入します。
10万円の車を買いました。
```

100万円の車を買いました。 — 引数が使われます

10万円の車を買いました。 — デフォルト値が使われます

このコードでは、2回buy()関数を呼び出しています。1度目の呼び出しでは引数として100を渡していますが、2度目には引数を渡していません。このため、2度目の呼び出しではデフォルト値の10が使われています。デフォルト引数を設定した場合には、実引数を指定しないで呼び出すことができるのです。

```
void buy(int x)
{
   cout << x << "万円の車を買いました。¥n";
}

int main()
{
   buy();
}
```


図7-15 デフォルト引数
関数にはデフォルト引数を指定しておくことができます。実引数を指定しないと、デフォルト値が使われます。

ただし、引数が複数ある場合には注意しなければなりません。デフォルト引数は、

右から順に設定しなければならない

ことになっています。つまり、ある引数のデフォルト値を設定した場合は、その引数以降にある引数もすべてデフォルト引数を設定しなければならないのです。

たとえば、5つの引数をもつ関数の場合には、func1()関数のように、右から順にデフォルト引数を指定することができます。func2()関数のように、2番目と5番目の引数にだけデフォルト値の設定をすることはできないのです。

右から順にデフォルト引数を設定することができます

```
//正しい
void func1(int a, int b, int c, int d=2, int e=10)

//誤り
//void func2(int a, int b=2, int c, int d, int e=10)
```

このようなデフォルト引数の指定はできません

このfunc1()関数は次のように呼び出すことができます。

① 2つの引数を省略して呼び出しています

```
func1(10, 5, 20);
func1(10, 5, 20, 30);
func1(10, 5, 20, 30, 50);
```

② 1つの引数（最後の引数）を省略して呼び出しています

③ すべての引数を指定して呼び出しています

①は、デフォルト引数を指定した2つの引数を省略して呼び出しています。つまり引数dは2、eは10で初期化されます。

②は、最後の引数を省略して呼び出しています。つまり、最後の引数eが10で初期化されます。

③は、すべての引数を指定して呼び出しています。

7.7 関数のオーバーロード

オーバーロードのしくみを知る

ここまで、いろいろな関数の定義をみてきました。これらの関数の中には、実質的には同じ処理をしようとするにもかかわらず、1つの関数にまとめられないものもあります。

たとえば、Sample10のmax()関数を思い出してください。このmax()関数は、「2つのint型の値の最大値を求める」という処理をしていました。しかし、int型ではなく、double型の最大値を求めたい場合はどうしたらいいのでしょうか？ この場合、double型の引数をもつ関数を新しく定義する必要があります。つまり、「最大値を求める」という関数を2つ用意しなければならないのです。

このように、実質的に同じ処理であっても、複数の関数を定義しなければならないことがあります。ただし、このときC++には便利な規則があります。

引数の型や数が異なっていれば、同じ名前をもつ関数を複数定義できる

のです。つまり、「最大値を求める関数」として、「max()関数」を2つ作ることができるのです。

```
int max(int x, int y)
double max(double x, double y)
```

int型を扱うmax()関数です

double型を扱うmax()関数です

このように、引数の数・型が異なる同じ名前の関数を複数定義することを、**関数のオーバーロード**（多重定義：function overloading）といいます。

それでは、関数のオーバーロードについてコードをみてみることにしましょう。

Sample12.cpp ▶ 関数をオーバーロードする

```
#include <iostream>
using namespace std;

//max関数の宣言
int max(int x, int y);
double max(double x, double y);

int main()
{
   int a, b;
   double da, db;

   cout << "2つの整数を入力してください。¥n";
   cin >> a >> b;

   cout << "2つの小数を入力してください。¥n";
   cin >> da >> db;

   int ans1 = max(a, b);
   double ans2 = max(da, db);

   cout << "整数値の最大値は" << ans1 << "です。¥n";
   cout << "小数値の最大値は" << ans2 << "です。¥n";

   return 0;
}

//max(int型)関数の定義
int max(int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
}

//max関数(double型)の定義
double max(double x, double y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

引数の異なる2つのmax()関数を宣言します

int型の引数をもつ関数が呼び出されます

double型の引数をもつ関数が呼び出されます

int型の引数をもつmax()関数です

double型の引数をもつmax()関数です

Sample12の実行画面

```
2つの整数を入力してください。
5 ↵
10 ↵
2つの小数を入力してください。
3.14 ↵
45.192 ↵
整数値の最大値は10です。
小数値の最大値は45.192です。
```

int型の引数をもつ関数による出力です

double型の引数をもつ関数による出力です

このコードでは、2種類のmax()関数を呼び出しています。このとき、

1番目に、引数がint型のもの
2番目に、引数がdouble型のもの

であるmax()関数が、それぞれ正しく呼び出されていますね。つまり、

似たような複数の処理をオーバーロードしておけば、1つの関数名をおぼえて使うだけで、自動的にその型・個数に応じた処理が行われる

ということになります。

関数の数が増えたとしても、オーバーロードの機能によって、わかりやすく、利用しやすいコードを記述することができます。

max
max(int x, int y)
max(double x, double y)

重要

同じ関数名で異なる引数の型・個数をもつ関数を定義できる。

```
int main()
{
    int ans1 = max(a, b);
    double ans2 = max(da, db);
}

int max(int x, int y)
{

}

double max(double x, double y)
{

}
```


図 7-16 **関数のオーバーロード**
関数をオーバーロードすると、呼び出しのときに渡す引数によって適切な関数が呼び出されます。

オーバーロードについての注意

ところで、最初に説明したように、オーバーロードする関数は、**引数の型や数が異なっていなければなりません。**

もし、引数の型と数がまったく同じで、戻り値だけが異なる次の2つの関数をオーバーロードできてしまうとしたら、どうなるのでしょうか？

```
int func(int a);
void func(int a);
```


すると、次のような呼び出しをしても、2つのうちどちらの関数を呼び出すものなのか、判断できないことでしょう。

```
func(10);
```


つまり、関数をオーバーロードするときには、引数の型または個数が異なるよ

うにしておかなければならないのです。

また、同じ理由で、デフォルト引数を使ったときにも、関数のオーバーロードができない場合があります。たとえば、次の関数をみてください。

```
int func(int a, int b=0);
int func(int a);
```

デフォルト引数を使っています

このとき、次の呼び出しによって、どちらの関数を呼び出すものなのか判断できません。

```
func(10);
```

どちらの関数を呼び出すべきなのか判断できません

オーバーロードの際には引数に注意する必要があるのです。

重要

オーバーロードする関数は、引数の型または個数が異なるようにする。

7.8 関数テンプレート

関数テンプレートのしくみを知る

関数のオーバーロードによって、同じ名前の関数を定義できることがわかりました。オーバーロードによって関数を利用しやすくなっていることが理解できるでしょう。

しかし、7.7節のmax()関数をもう一度みてください。扱っている型以外は、まったく同じ処理が行われているにもかかわらず、複数の関数を定義しなければなりません。これでは、関数の扱いにまだまだ面倒な部分があるような気がしますね。そこで、この節ではさらに関数を便利に使う方法を紹介しましょう。

C++では、

関数の"ひながた"を用意できる

という便利な機能があります。扱う型だけが異なる関数を、ひながたから作成できるようになっているのです。このひながたのことを、**関数テンプレート**（function template）と呼びます。関数テンプレートを利用するには、次のような手順が必要です。

1. 関数テンプレートを宣言・定義する
2. 関数を呼び出す（関数が自動的に生成される）

まずはじめに、関数テンプレートを宣言・定義する方法をみてみましょう。

関数テンプレートを定義する

関数テンプレートの宣言・定義は次のように記述します。ちょっとややこしいですが、かたちをよく頭に入れてください。

構文　関数テンプレートの宣言・定義

```
template <classテンプレート引数のリスト>
  関数の宣言または定義
```

置きかえる型名を指定します

関数テンプレートは、通常の関数宣言または定義にtemplate< ･･･ >という指定をつけたものです。< ･･･ >の部分には、**テンプレート引数**というものを指定します。

テンプレート引数は、Tなどという仮の型名を指定したものです。関数テンプレートでは、仮引数の型名に、テンプレート引数を使います。具体的な型名のかわりに、Tなどという仮の型名を使うのです。

次のコードが、関数テンプレートの定義です。

```
//関数テンプレート
template <class T>
T maxt(T x, T y)
{
   if(x > y)
      return x;
   else
      return y;
}
```

Tなどと記述しておきます

具体的な型名のかわりにTを仮の型名とします

```
template <class T>
T maxt(T x, T y)
{
   if(x > y)
      return x;
   else
      return y;
}
```

図7-17　関数テンプレートの定義
関数テンプレートは、関数の「ひながた」となります。

関数テンプレートを利用する

それでは関数テンプレートを利用してみましょう。関数テンプレートを利用するには、通常の関数の呼び出しと同じように、関数を呼び出します。関数テンプレートの呼び出しを記述すると、コードをコンパイルするときに、

テンプレート引数Tを、指定した型に置きかえた関数が作成される

ということになっています。つまり、コードで関数テンプレートを呼び出すことによって、具体的な型を扱う関数が呼び出されるのです。

それでは、関数テンプレートを使って、7.6節と同じ処理を行うコードを記述してみましょう。

Sample13.cpp ▶ 関数テンプレートを使う

```
#include <iostream>
using namespace std;

//関数テンプレート
template <class T>
T maxt(T x, T y)
{
   if(x > y)
      return x;
   else
      return y;
}

int main()
{
   int a, b;
   double da, db;

   cout << "2つの整数を入力してください。¥n";
   cin >> a >> b;

   cout << "2つの小数を入力してください。¥n";
   cin >> da >> db;

   int ans1 = max(a, b);
```

テンプレート引数です

テンプレート引数をint型に置きかえた関数が呼び出されます

```
    double ans2 = max(da, db);

    cout << "整数値の最大値は" << ans1 << "です。¥n";
    cout << "小数値の最大値は" << ans2 << "です。¥n";

    return 0;
}
```

テンプレート引数をdouble型に置きかえた関数が呼び出されます

ここでは、まずint型の値を渡して関数テンプレートを呼び出しています。次にdouble型の値を渡してもう一度関数テンプレートを呼び出しています。このような呼び出しによって、テンプレート引数Tを、int型、double型で置きかえた関数が作成されます。そして、プログラムを実行するときには、指定した型の関数が呼び出されるようになっているのです。

関数テンプレートは、

扱う型は異なるが、そのほかではまったく同じ処理を行う関数について、コードを1つにまとめることができる

という便利な機能をもっています。2つのmax()関数は、型以外の処理がまったく

図7-18 関数テンプレートの利用
関数テンプレートに実引数を渡すと、その型を扱う関数が作成されます。

Lesson 7

同じだったので、この関数テンプレートを使うことができるのです。

関数を便利に利用する

この章で紹介した「関数のオーバーロード」、「関数テンプレート」は、異なる複数の処理を、同じ関数名を使って利用できるようにした機能だといえます。このように、1つの名前が状況に応じて別々のはたらきをもつことを、**多態性**（ポリモーフィズム：Polymorphism）といいます。ただし、オーバーロードとテンプレートでは、使える局面が異なります。それぞれの特徴に注意しておきましょう。

関数のオーバーロード	関数内の処理手続きが異なってもよい
関数テンプレート	関数内の処理手続きはまったく同じである必要がある。扱う型だけが異なる場合に使うことができる

7.9 レッスンのまとめ

この章では、次のようなことを学びました。

- 一定の処理をまとめて関数として定義し、呼び出すことができます。
- 関数の本体に引数を渡して処理させることができます。
- 関数の本体から戻り値を受けとることができます。
- かんたんな処理を行う関数は、インライン関数とすることができます。
- 関数プロトタイプ宣言によって、関数の仕様をコンパイラに知らせることができます。
- 引数にはデフォルト値を指定できます。
- 同じ名前の関数を複数定義する（オーバーロードする）ことができます。
- 関数テンプレートから、扱う型のみが異なる関数を作成することができます。

C++のプログラムを作成するには関数が欠かせません。適切な処理を関数としてまとめ、関数を呼び出すことによって、複雑なコードをよりかんたんに記述できるようになります。またこのあとの第10章では、関数をほかのプログラムで再利用する方法も学びます。

練習

1. int型の数値を２乗した値を返す関数int square(int x)を作成し、次のようにキーボードから入力した整数の２乗を出力するコードを記述してください。

```
整数を入力してください。
5 ↵
5の2乗は25です。
```

2. １のコードに、double型の数値を２乗する関数double square(double x)を追加してください。次のようにキーボードから入力した整数・小数の２乗を出力するコードを追加してください。

```
整数を入力してください。
5 ↵
5の2乗は25です。
小数を入力してください。
1.5 ↵
1.5の2乗は2.25です。
```

3. ２のコードで、2つの関数をインライン関数として記述してください。

4. 与えられた型の数値を２乗する関数テンプレートtemplate <class T>T squaret(T x)を作成してください。int型、double型の数値を入力して、２乗を出力するコードを記述してください。

Lesson 8

ポインタ

第3章では、変数を使って値を記憶する方法について学びました。C++には、メモリ上の位置を直接あらわすために、「ポインタ」という機能があります。ポインタを理解するには、コンピュータのメモリの概念を理解することが欠かせません。むずかしい部分もありますが、しっかりと学んでいきましょう。この章では、ポインタの意味とその使いかたを説明します。

Check Point!

- メモリ
- アドレス
- ポインタ
- アドレス演算子＆
- 間接参照演算子＊
- 参照
- 実引数の変更

8.1 アドレス

アドレスのしくみを知る

第3章では、変数の値がコンピュータの「メモリ」に記憶されることを学びました。C++には、メモリの位置を直接あらわす「ポインタ」という機能が用意されています。ポインタはなかなかわかりにくいしくみですが、一歩ずつしっかりと身につけていきましょう。

この章ではまずはじめに、メモリ上の位置を直接あらわす**アドレス**（address：番地）という言葉から学んでいくことにします。

「アドレス」と聞いて、皆さんは、何を思いうかべるでしょうか？ 家の「住所」や、メールのアドレスなどを思いうかべるかもしれません。

C++の「アドレス」とは、メモリの場所を直接あらわすために使われる、メモリ上の「住所」のことです。コンピュータ内の住所らしく、16進数の数値などを使って、0x1000、0x1004・・・という数値であらわすことが多くなっています。

図9-1 メモリ・変数・アドレス
アドレスはメモリ上の「位置」を直接あらわすために使われます。

変数のアドレスをみる

しかし、「メモリ」や「アドレス」といっても、ピンとこない方もいるかもしれません。そこで、「アドレス」の値を実際にみてみることにしましょう。

変数の値が格納されているメモリのアドレスを知るには、**アドレス演算子**（address operator）の**&**を使います。

構文　アドレス演算子

```
&変数名
```

変数のアドレスをあらわします

それでは、&演算子を使って、アドレスを実際に出力してみましょう。次のコードを入力してみてください。

Sample1.cpp ▶ アドレスを出力する

```
#include <iostream>
using namespace std;

int main()
{
   int a;

   a = 5;

   cout << "変数aの値は" << a << "です。¥n";
   cout << "変数aのアドレスは" << &a << "です。¥n";

   return 0;
}
```

変数aのアドレスを出力します

Sample1の実行画面

```
変数aの値は5です。
変数aのアドレスは0x00F4です。
```

変数aのアドレスが出力されます

最初の行ではこれまでどおり、変数aの値である5が出力されていますね。次に、このコードでは、アドレス演算子を使った「&a」を出力しています。これで変数aのアドレスを出力することができます。「&a」によって、

変数aの値がメモリ上の「どの位置」に記憶されているか

を知ることができるのです。

&aの値をみると、「0x00F4」(16進数)と出力されていることがわかります。これが変数aの値が格納されているメモリの場所を意味する「アドレス」なのです。このコンピュータの場合、変数aの値は図のような0x00F4というメモリの位置に格納されていることがわかります。

なお、ここでは0x00F4という値が出力されましたが、アドレスの値は、使っている環境やプログラムの実行状況によってかわってきます。ほかのコンピュータでは、0x00F4ではなく、別の値が出力されることでしょう。

しかし、変数のアドレスが実際にいくつであるかというのは、あまり意味のあることではありません。ここで大事なことは、

アドレスを使って、メモリ上の「位置」をあらわすことができる

ということなのです。これはしっかりとおぼえてください。

重要

メモリ上の位置を示すためにアドレスが使われる。

&a
アドレス
0x00F4
メモリ
変数
a

図8-2 アドレス演算子
変数名に&演算子をつけると、その変数のアドレスを知ることができます。

8.2 ポインタ

ポインタのしくみを知る

さて、変数の値が格納されているメモリの場所（アドレス）を知ることができたので、これを利用していくことにしましょう。

まず、アドレスを使うコードを記述するために、

アドレスを格納する特殊な変数

を学ぶことにします。この変数は**ポインタ**（pointer）と呼ばれています。

ポインタの使いかたは、原則としてこれまでの変数と同じです。第3章の変数と同じように、ポインタも使う前に「pA」などという名前を決めて宣言します。ただし、ポインタをあらわす変数には、必ず*という記号をつけて宣言することになっています。次のポインタの宣言方法をみてください。

図8-3 ポインタ
ポインタはアドレスを格納することができる変数です。

構文 ポインタの宣言

```
型名* ポインタ名;
```
ポインタを宣言します

つまり、ポインタの宣言は次のように行うことになります。

```
int* pA;
```
ポインタpAを宣言します

この文は、

int型の変数のアドレスを格納できるポインタpA

というものを宣言したものです。これを**int型へのポインタpA**ということもあります。かわったいいかたですが、頭のすみにおいておくことにしましょう。

ポインタには、原則として指定した型以外の値のアドレスは格納できません。つまり、pAにはint型以外の値のアドレスを格納することはできません。型名を指定するときには注意してください。

それでは、ポインタpAに何かアドレスを格納してみることにします。

Sample2.cpp ▶ ポインタにアドレスを格納する

```
#include <iostream>
using namespace std;

int main()
{
   int a;
   int* pA;    ← ① ポインタpAを宣言します

   a = 5;
   pA = &a;    ← ② 変数aのアドレスをpAに格納します

   cout << "変数aの値は" << a << "です。¥n";
   cout << "変数aのアドレス(&a)は" << &a << "です。¥n";
   cout << "ポインタpAの値は" << pA << "です。¥n";
                                  ↑ ③ pAの値（変数aのアドレス）を出力します
   return 0;
}
```

Sample2の実行画面

```
変数aの値は5です。
変数aのアドレス(&a)は0x00F4です。
ポインタpAの値は0x00F4です。    ← ポインタの内容は変数aのアドレスです
```

さきほどみたように、&aはint型の変数aのアドレスをあらわしています。そこで、この&aという値を②のようにポインタpAに代入することができます。

```
pA = &a;
```
変数aのアドレスをpAに格納します

つまり、この代入によって、

ポインタpAに変数aのアドレスを格納する

という作業ができました。このため、ポインタpAの値として出力される値は、変数aのアドレスである、&aの値と同じとなっています。つまり、②のような代入によって、

変数aとポインタpAの間には何かしらの「関係」ができた

とみることもできるでしょう。この「関係」のことを、

pAは変数aをさす

などといったりします。不思議ないいかたですが、ポインタpAには変数aのメモリ上の位置（アドレス）が入っていることを思い出してください。

pA（の値）が、変数a（の場所）をさし示すようになった

と考えればよいのです。

図8-4 変数とポインタ
① int型の変数aに5を代入します。
② int*型のポインタpAに変数aのアドレスを代入します。

重要

ポインタにアドレスを格納することができる。

ポインタから変数の値を知る

実は、ポインタに変数のアドレスが格納されると、

そのポインタから逆にたどって、元の変数の値を知ることができる

ということができるようになります。ポインタから変数の値をたどるには、ポインタに対して、*という演算子を使います。*演算子は**間接参照演算子**（indirection operator）と呼ばれています。

構文 **間接参照演算子**

```
*ポインタ名;
```

この演算子を使うと、そのポインタに格納されているアドレスに対応する変数の値を知ることができます。

たとえば、ポインタpAに変数aのアドレスが格納されている場合には、

```
*pA
```

と記述すれば、**変数aの値を「間接的」に知ることができる**わけです。

では、次のコードでたしかめてみることにしましょう。

Sample3.cpp ▶ 間接参照する

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int a;
    int* pA;

    a = 5;
    pA = &a;

    cout << "変数aの値は" << a << "です。¥n";
    cout << "変数aのアドレスは" << &a << "です。¥n";
    cout << "ポインタpAの値は" << pA << "です。¥n";
    cout << "*pAの値は" << *pA << "です。¥n";

    return 0;
}
```

変数aのアドレスをpAに格納します

*を使うと、ポインタがさしている変数の値がわかります

Sample3の実行画面

```
変数aの値は5です。
変数aのアドレスは0x00F4です。
ポインタpAの値は0x00F4です。
*pAの値は5です。
```

ポインタがさしている変数の値が出力されます

このコードでも、最初にポインタpAに変数aのアドレスを代入しています。つまり、ポインタpAが変数aをさすようにするわけです。そのあとでpAに*演算子を使うと、変数aの値を知ることができます。pAに*をつけた「*pA」は、変数aと同じものをあらわすことに注目してください。つまり、

*pA ←→ a
同じ

となっているのです。たしかに、*pAの値を出力してみると、変数aの値である「5」が出力されているのがわかりますね。

重要

間接参照演算子*によって、ポインタがさしている変数の値を知ることができる。

Lesson 8

図8-5　間接参照演算子
ポインタに * 演算子を使うと、ポインタがさしている変数の値を知ることができます。

ポインタについて整理する

かなりややこしくなってきたので、いままで出てきたことがらを順を追って、整理してみましょう。まず、変数aとそのアドレス&aは次のようにあらわされています。

a	変数a
&a	変数aのアドレス

ここで、「pA = &a;」と代入します。つまり、ポインタpAが変数aをさすようにするわけです。すると、pAと*pAは次のようになります。

pA	変数aのアドレスを格納したポインタ
*pA	変数aのアドレスを格納するポインタがさしている変数 ➡ 変数a

最後のほうになってくるとまわりくどくていやな感じですが、ここが最も肝心なところです。「pA = &a;」という代入をしなければ、後半の2つはなりたたないことに注意してください。

ポインタの宣言の復習

この章で学んだ、ポインタの宣言を思い出してください。

```
int* pA;
```

という宣言は、int型の変数のアドレスを格納するポインタpAを準備するものでした。このことを、

ポインタpAはint*型である

ともいいます。

実は、ポインタを宣言する際には、次のように書くこともできるようになっています。

```
int* pA;
int *pA;
```

この文はどちらも同じことをあらわしています。*の意味から考えてみれば、下は、

***pAはint型である**

ということもあらわしているということになります。ただし、本書では、int* pAの表記を使うことにしましょう。

また、複数のポインタを , で区切って宣言する際には注意が必要です。

```
int* pA, pB;
```

という宣言は、pAとpBの両方をint*型としたようにみえますが、これは、

```
int* pA;
int pB;
```

という2つの宣言と同じになります。どちらもint*型としたい場合には、カンマを

使わずに複数行で宣言するか、

```
int *pA, *pB;
```

とする必要があります。

ポインタに別のアドレスを代入する

これまで説明してきたように、ポインタはアドレスを格納する変数となっています。そこで今度は、変数aではなく、別の変数bのアドレスをpAに格納するように、ポインタの値を変更してみましょう。

Sample4.cpp ▶ ポインタの値を変更する

```
#include <iostream>
using namespace std;

int main()
{
   int a = 5;
   int b = 10;
   int* pA;

   pA = &a; ●─── 変数aのアドレスを代入します

   cout << "変数aの値は" << a << "です。¥n";
   cout << "ポインタpAの値は" << pA << "です。¥n";
   cout << "*pAの値は" << *pA << "です。¥n";

   pA = &b; ●─── 変数bのアドレスを代入します

   cout << "変数bの値は" << b << "です。¥n";
   cout << "ポインタpAの値は" << pA << "に変更されました。¥n";
   cout << "*pAの値は" << *pA << "です。¥n";

   return 0;
```

```
}
```

Sample4の実行画面

```
変数aの値は5です。
ポインタpAの値は0x00F4です。
*pAの値は5です。
変数bの値は10です。
ポインタpAの値は0x00F0に変更されました。
*pAの値は10です。
```


まず、ポインタpAに変数aのアドレスを代入しています。このため、*pAの値を出力してみると、変数aの値と同じ「5」が出力されますね。これはいままでと同じです。

その次に、「pA = &b;」と代入して、ポインタの値を変更してみました。今度はポインタpAに変数bのアドレスが格納されたのです。

もう一度*pAを出力してみると、今度は変数bの値と同じ「10」が出力されているのがわかります。つまり、ポインタpAが変数bをさすように変更できたわけです。このように、ポインタの値を変更し、別の変数をさすようにすることができます。

重要

ポインタにはさまざまな変数のアドレスを格納できる。

図8-6 **ほかの変数のアドレスを代入する**
ポインタにほかの変数のアドレスを代入して、値を変更することができます。

なお、ポインタには、指定した型の値のアドレス以外は原則として格納できません。pAの場合には、int型以外の値のアドレスを格納することができないので、注意しておきましょう。

ポインタに代入をしないと？

ポインタを扱うには注意しておかなければならないことがあります。「pA = &a;」という代入をしないで、*pAを出力した場合のことを考えてみてください。

```
//このコードは誤り
int a = 5;
int* pA;
```

pAには代入をしていません

```
cout << "ポインタpAの値は" << pA << "です。¥n";
cout << "*pAの値は" << *pA << "です。¥n";
...
```

どこをさしているのかわかりません

ここには、「pA = &a;」という文がありません。つまり、ポインタpAにはどのアドレスも格納されていません。**ポインタが何もさし示していない状態になっている**のです。これでは*pAと記述しても、意味のある値を得ることができません。

pA
?
?
int*型

このように、どこをさし示しているかわからないポインタを使うと、プログラムの実行時に思いもかけないエラーが生じてしまうことがあります。

ポインタは、できる限り次のように初期化を行うように記述したほうがよいでしょう。


```
int a;
int* pA = &a;
```

ポインタを初期化します

このように宣言時に変数のアドレスで初期化するようにすれば、うっかりポインタに値を代入するのを忘れて、ポインタが何もさしていない状態にならずにすみます。

重要

ポインタには必ずアドレスを代入して使用する。

ポインタを使って変数を変更する

さて、ポインタをさらに利用してみましょう。実はポインタを使うと、ポインタがさしている変数そのものの値を変更することができるようになっています。次

のコードをみてください。

Sample5.cpp ▶ ポインタを使って変数の値を変更する

```
#include <iostream>
using namespace std;

int main()
{
    int a;
    int* pA;

    a = 5;
    pA = &a;

    cout << "変数aの値は" << a << "です。¥n";

    *pA = 50;

    cout << "*pAに50を代入しました。¥n";
    cout << "変数aの値は" << a << "です。¥n";

    return 0;
}
```

*pA、つまり変数aに値を代入しました

変数aの値を出力してみると・・・

Sample5の実行画面

```
変数aの値は5です。
*pAに50を代入しました。
変数aの値は50です。
```

変数aの値が変更されています

このコードでは変数aの値が途中で変更されていますね。しかし、これは第3章のように、「a = 50;」という代入文を記述して、変数aを変更したのではありません。「*pA = 50;」というコードを記述したのです。これは、ポインタpAがaをさすとき、*pAとaが同じものを意味することを利用したものです。つまり、

*pA ⟷ a
同じ

ですから、

```
*pA = 50;  ←→  a = 50;
           同じ
```

といえますね。「a = 50;」と「*pA = 50;」 のどちらも変数aに値を格納する処理を行う文だといえるわけです。

しかし、変数aの値を変更する目的ならば、*pAに値を代入するよりもaに直接代入するほうがずっとかんたんな気がします。なぜ、こんなまわりくどい方法が用意されているのでしょうか？ 次の節では、このしくみを利用してポインタを使いこなす方法をみることにしましょう。

重要

間接参照演算子*を使って、ポインタがさす変数に値を代入できる。

図8-7　*pAに値を代入する
「*pAに値を代入する」ことは、「ポインタpAがさしている変数aに値を代入する」ことだといえます。

8.3 引数とポインタ

動作しない関数

まず、ポインタの便利な使いかたを学ぶ前に、第7章で学んだ「関数」を思い出してください。この章では次のswap()という名前の関数を定義してみることにしましょう。

```
//誤ったswap関数の定義
void swap(int x, int y)
{
   int tmp;

   tmp = x;
   x = y;
   y = tmp;
}
```

xとyを交換するつもりの関数です①～③

swap()関数は、引数xとyを交換する処理を行う関数として作成しました。この関数内では、次のような手順で変数xとyの値を交換しています。

① xの値をtmpに代入する
↓
② yの値をxに代入する
↓
③ tmpの値（元のxの値）をyに代入する

変数tmpを介してxとyを交換しているわけです。これでxの値とyの値が交換されるはずですね。

ところが、この関数を呼び出してみると、実際には、思ったとおりの処理が行われないのです。swap()関数を実際に使ってみましょう。

Sample6.cpp ▶ 誤った関数を使ってみる

```
#include <iostream>
using namespace std;

//誤まったswap関数の宣言
void swap(int x, int y);

int main()
{
    int num1 = 5;
    int num2 = 10;

    cout << "変数num1の値は" << num1 << "です。¥n";
    cout << "変数num2の値は" << num2 << "です。¥n";
    cout << "変数num1とnum2の値を交換します。¥n";

    swap(num1, num2);  ● swap()関数を呼び出しましたが・・・

    cout << "変数num1の値は" << num1 << "です。¥n";
    cout << "変数num2の値は" << num2 << "です。¥n";

    return 0;
}

//誤ったswap関数の定義
void swap(int x, int y)
{
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}
```

Sample6の実行画面

```
変数num1の値は5です。
変数num2の値は10です。
変数num1とnum2の値を交換します。
変数num1の値は5です。
変数num2の値は10です。
```

値は交換されませんでした

変数num1とnum2を交換するため、swap()関数を呼び出してnum1とnum2を実引数として渡しました。しかし、実行画面をみると、変数num1と変数num2の値は交換されていません。いったいどうしてなのでしょうか？

値渡しと参照渡し

この秘密をとくには、関数に引数を渡す方法について、第7章をもう一度復習しておく必要があります。関数に実引数を渡す際には、

実引数の「値」だけが関数内に渡される

ということになっています。このような引数の渡しかたを**値渡し**（pass by value）といいます。たとえば、swap()関数では、変数num1とnum2の値である「5」と「10」だけが関数に渡されることになります。

```
int main()
{

   swap(num1, num2);

}

void swap(int x, int y)
{
   int tmp;

   tmp = x;
   x = y;
   y = tmp;
}
```

図8-8 関数の呼び出し（値渡し）
通常の値渡しの方法では、呼び出し元の実引数を変更することはできません。

図8-8をよくみてください。関数内で仮引数xとyの値を交換する処理を行っても、これは、変数num1とnum2の値を「コピー」した5と10を交換しているにすぎません。つまり、swap()関数内で値を交換しても、呼び出し元の変数であるnum1とnum2に影響を与えることができないのです。

ところがポインタをうまく使うと、関数の呼び出し時に指定した引数の値を変更することができるようになります。

このためには、swap()関数の**仮引数をポインタとして定義すること**が必要です。そこで、次のようにswap()関数を書きかえてみてください。

```
//swap関数の定義
void swap(int* pX, int* pY)
{
   int tmp;

   tmp = *pX;
   *pX = *pY;
   *pY = tmp;
}
```

今度は、仮引数にポインタを使いました。ただし、関数内では*を使ってポインタがさし示しているものどうしを交換しているので、さきほどのswap()関数と内容としては同じものとなっています。この関数を呼び出すには、仮引数がポインタなので、変数のアドレスを引数として渡します。さて、今度はどうなるでしょうか？

Sample7.cpp ▶ 関数の引数にポインタを使う

```
#include <iostream>
using namespace std;

//swap関数の宣言
void swap(int* pX, int* pY);

int main()
{
    int num1 = 5;
    int num2 = 10;

    cout << "変数num1の値は" << num1 << "です。¥n";
    cout << "変数num2の値は" << num2 << "です。¥n";
    cout << "変数num1とnum2の値を交換します。¥n";

    swap(&num1, &num2);   ● アドレスを渡して新しいswap()関数を呼び出します

    cout << "変数num1の値は" << num1 << "です。¥n";
    cout << "変数num2の値は" << num2 << "です。¥n";

    return 0;
}

//swap関数の定義
void swap(int* pX, int* pY)  ┐
{                            │
    int tmp;                 │
                             ├ 引数にポインタを使った関数の定義です
    tmp = *pX;               │
    *pX = *pY;               │
    *pY = tmp;               ┘
}
```

Sample7の実行画面

```
変数num1の値は5です。
変数num2の値は10です。
変数num1とnum2の値を交換します。
変数num1の値は10です。
変数num2の値は5です。
```

今度は正しく交換されました

呼び出すときに、変数num1とnum2のアドレス（&num1、&num2）を渡しました。すると、今度は無事にnum1とnum2が交換されていることがわかります。

このように、呼び出し元の変数の値を関数内で変更したい場合は、仮引数をポインタとしておきます。そして、関数を呼び出すときに、アドレスを関数に渡すことにします。すると、仮引数は渡されたアドレスで初期化されます。つまり、num1とnum2のアドレスが、ポインタpX、pYに格納されることになります。

仮引数		実引数
pX	←	num1のアドレス
pY	←	num2のアドレス

8.2節で説明したように、ポインタに*をつけたものは、そのポインタがさしている変数と同じものをあらわすのでした。つまり、

***pXや*pYはnum1、num2と同じものをあらわす**

と考えることができるようになります。8.2節と同じように書いてみると、ここでは、

***pX（仮引数側）⟷ num1（実引数側）**
同じ

***pY（仮引数側）⟷ num2（実引数側）**
同じ

という関係ができたわけです。

ですから、この関数内では、

仮引数側の*pX、*pYを変更する ⟷ 実引数側のnum1、num2を変更する
同じ

Lesson 8

ということになります。

図8-9をみてください。今度は、仮引数側と実引数側には特定の関係が存在しています。これで、関数の呼び出しによって呼び出し元の変数の値を交換できることになります。さきほどの関数では、仮引数に渡されるのは実引数の値のコピーであって、仮引数側と実引数側には何の関係も存在していませんでした。

このように、ポインタを関数の仮引数として使うと、実引数側の値を変更することができるようになります。

なお、関数が呼び出されるとき、実引数のアドレスが関数に渡されることを、**参照渡し**（pass by reference）と呼んでいます。ここで紹介したポインタを使った関数は、アドレスを指定して関数に渡すため、実質的に参照渡しの関数となっています。

```
int main()
{

   swap(&num1, &num2);

}
```

```
void swap(int* pX, int* pY)
{
   int tmp;

   tmp = *pX;
   *pX = *pY;
   *pY = tmp;
}
```


図8-9 関数の呼び出し（ポインタ）
仮引数にポインタを使うと、実引数側を変更することができます。

重要

原則として、関数内で実引数を変更することはできない。
ポインタを使うと、関数内で実引数側を変更することができる。

8.4 引数と参照

参照のしくみを知る

ポインタを関数の引数として利用すると、呼び出し元の変数を変更することができます。しかし、ポインタはメモリ上のアドレスを直接扱うため、むずかしい面もあります。

そこで、この節ではさらに新しい機能を学びましょう。**参照**（reference）という機能を学ぶことにします。

まず、参照について基本的なことがらを学んでおきましょう。参照は、変数などで初期化した識別子のことをいいます。型名に&をつけて宣言します。

Lesson 8

構文 **参照**

```
型名& 参照名 = 変数;
```

参照名には識別子を使います

構文だけではわかりくいので、実際に、参照を使ってみることにしましょう。

```
int a;
int& rA = a;
```

参照rAを変数aで初期化します

このコードの「rA」が参照です。rAを変数aで初期化したのです。参照を使ったかんたんな次の例をみてください。

Sample8.cpp ▶ 参照を使う

```
#include <iostream>
using namespace std;
```

```
int main()
{
   int a = 5;
   int& rA = a;           ● 参照rAを変数aで初期化しました

   cout << "変数aの値は" << a << "です。¥n";
   cout << "参照rAの値は" << rA << "です。¥n";

   rA = 50;               ● 参照rAに値を代入しました

   cout << "rAに50を代入しました。¥n";
   cout << "参照rAの値は" << rA << "に変更されました。¥n";
   cout << "変数aの値も" << a << "に変更されました。¥n";
   cout << "変数aのアドレスは" << &a << "です。¥n";
   cout << "参照rAのアドレスも" << &rA << "です。¥n";

   return 0;
}
```

Sample8の実行画面

```
変数aの値は5です。
参照rAの値は5です。
rAに50を代入しました。
参照rAの値は50に変更されました。
変数aの値も50に変更されました。
変数aのアドレスは0x00F4です。
参照rAのアドレスも0x00F4です。
```

変数aと参照rAの値は同じです

参照rAに値を代入すると変数aの値が変更されます

参照rAのアドレスは変数aと同じなのです

上の出力結果をみるとわかるように、参照rAの値やアドレスは、変数aとまったく同じです。参照rAを変数aで初期化することによって、rAはaと同じものであると指定しているのです。つまり、

rA ⟷ a
同じ

となるわけです。そこで、ポインタのときと同じように考えてみましょう。このことから、

rA = 50; ←→ a = 50;
同じ

ということがいえます。つまり、Sample8のように、参照rAを使うと変数aの値を変更することができるわけです。参照rAを使って、変数aをとり扱うことができるのです。

図8-10 参照
参照rAを使って、rAを初期化した変数aを扱うことができます。

ただし、参照は必ず参照の対象となる変数で初期化しておかなければなりません。つまり、

```
int& rA;
rA = a;
```

参照に代入することはできません

というように、あとから代入することはできません。

引数に参照を使う

さて、これで参照を使って、8.3節のswap()関数をポインタを使わずに定義することができそうですね。参照を使って、新しいswap()関数を記述してみましょう。

Sample9.cpp ▶ 関数の引数に参照を使う

```
#include <iostream>
using namespace std;

//swap関数の宣言
void swap(int& x, int& y);

int main()
{
   int num1 = 5;
   int num2 = 10;

   cout << "変数num1の値は" << num1 << "です。¥n";
   cout << "変数num2の値は" << num2 << "です。¥n";
   cout << "変数num1とnum2の値を交換します。¥n";

   swap(num1, num2);

   cout << "変数num1の値は" << num1 << "です。¥n";
   cout << "変数num2の値は" << num2 << "です。¥n";

   return 0;
}

//swap関数の定義
void swap(int& x, int& y)
{
   int tmp;

   tmp = x;
   x = y;
   y = tmp;
}
```

Sample9の実行画面

```
変数num1の値は5です。
変数num2の値は10です。
変数num1とnum2の値を交換します。
変数num1の値は10です。
変数num2の値は5です。
```

正しく交換されています

参照を引数としたswap()関数を使うと、変数num1と変数num2をそのまま実引数として渡すことができます。つまり、参照である仮引数（x、y）は実引数（num1、num2）で初期化されます。この初期化は、

x（仮引数側） ⟷ num1（実引数側）
同じ
y（仮引数側） ⟷ num2（実引数側）
同じ

を意味するわけです。図8-11をみてください。ここでは、仮引数を交換すると実引数も交換されることが示されています。したがって参照を使えば、ポインタを引数としたときと同じように、仮引数を変更すると実引数の値を変更することができます。


```
int main()
{
   swap(num1, num2);
}

void swap(int& x, int& y)
{
   int tmp;

   tmp = x;
   x = y;
   y = tmp;
}
```


図8-11 関数の呼び出し（参照）
仮引数に参照を使うと、実引数側を変更できます。

このように、参照はポインタと同じようなはたらきをもっています。しかし、参照はポインタとは異なる概念なので、ポインタに参照を代入するなどといったことはできません。

重要

関数内で実引数を変更する場合には、参照を使う。

実引数を変更したくない場合には？

ところで、ポインタや参照を使っても、必ずしも実引数を変更する必要はありません。関数の内容によっては、ポインタや参照を使っていても、実引数を変更したくない場合もあります。

このようなときには、関数内で実引数の値を変更しないことをはっきりと示すために、仮引数にconstという指定をつけることもあります。

```
void func1(const int* pX);
void func2(const int& X);
```

constをつけると、関数内では実引数の変更ができなくなります

上のコードは、実引数を変更しないことを明示した関数プロトタイプ宣言です。constを使った場合、関数内で引数の変更をする処理を記述すると、エラーとなります。

```
//func1関数の定義
void func1(const int* pX)
{
   cout << *pX << "を出力します。¥n";

  //*pX = 10;
}
void func2(const int& x)
{
   cout << x << "を出力します。¥n";

  //x = 10;
}
```

constをつけると・・・

実引数を変更できなくなります

上のfunc1()関数、func2()関数内では、引数を変更する処理を行うことはできません。

constは指定してもしなくてもかまいません。しかし、指定することで、実引数を変更しないことをはっきり示すことができ、誤りのおきにくいコードを記述することができます。

重要

constを指定した引数は、関数内で値を変更することはできない。

8.5 レッスンのまとめ

この章では、次のようなことを学びました。

- アドレスは、メモリ上の位置を直接あらわします。
- ポインタは、特定のアドレスを格納する変数です。
- アドレス演算子（&）を使うと、変数などのアドレスを知ることができます。
- ポインタに間接参照演算子（*）を使うと、ポインタがさしている変数の値を得ることができます。
- 引数は原則として値渡しで関数に渡されます。
- 関数の仮引数にポインタを使うと、呼び出し元の実引数を変更することができます。
- 関数の仮引数に参照を使うと、呼び出し元の実引数を変更することができます。
- 仮引数にconstを指定すると、実引数を変更することができなくなります。

ポインタを使うと、メモリ上の位置を直接あらわすことができます。ポインタを使って、関数の引数を参照渡しとする方法も学びました。ポインタの概念はややこしいものですが、あせらずにじっくりとひとつずつ復習してみてください。

練習

1. 次の項目について○か×で答えてください。

① 次のコードの記述によって、どのコンピュータでも同じように出力される。

```
#include <iostream>
using namespace std;

int main()
{
    int a = 5;
    cout << "変数aのアドレスは" << &a << "です。¥n";

    return 0;
}
```

② ポインタを宣言したあと、そのポインタに別の文で変数のアドレスを代入することができる。

③ 参照を宣言したあと、その参照に別の文で変数を代入することができる。

2. 2科目のテストの点数(x1, x2)に、a点を加算する関数add()関数を、ポインタを使って定義してください。キーボードからx1、x2、aを入力し、次のように出力するコードを記述してください。

```
2科目分の点数を入力してください。
78 ↵
65 ↵
加算する点数を入力してください。
12 ↵
12点加算しましたので
科目1は90点となりました。
科目2は77点となりました。
```

3. 2と同じ処理を行う関数を、参照を使って定義してください。

Lesson 9

配列

第3章では、変数を使って特定の値を記憶するしくみについて学びました。さらにC++には、同じ型の複数の値をまとめて記憶する「配列」という機能があります。配列を使うと、たくさんのデータを処理する複雑なコードを、すっきりと記述できるようになります。この章では、配列のしくみについて学んでいくことにしましょう。

Check Point!

- 配列
- 配列の宣言
- 配列要素
- 添字
- 配列の初期化
- 配列とポインタの関係
- ポインタの演算
- 引数と配列
- 文字列と配列
- 標準ライブラリ

9.1 配列

配列のしくみを知る

プログラムの中では、たくさんのデータを扱う場合があります。たとえば、50人の学生がいるクラスのテストの点数を扱うプログラムを考えてみてください。

これまでに学んできた知識を使うと、50人のテストの点数を変数に記憶させて管理するコードを書くことができます。そこで、test1からtest50という名前の変数を全部で50個用意してみることにしましょう。

```
int test1 = 80;
int test2 = 60;
int test3 = 22;
...
int test50 = 35;
```

50個の変数を初期化しています

ただし、こうしてたくさんの変数が登場すると、コードが複雑で読みにくくなってしまう場合があります

このようなときには、**配列**（array）というしくみを利用すると便利です。

変数が、特定の1つの値を記憶させる機能をもっていたことを思い出してください。配列も「特定の値を記憶する」という点で、変数と同じ役割をもっています。ただし、配列は、

同じ型の値を複数まとめて記憶する

という便利な機能を備えています。

いくつもの同じ名前のついたハコが、組になって並んでいるイメージを思いうかべてみてください。変数と同じように、配列のハコの中にも値を格納して利用することができます。配列のそれぞれのハコは、配列の**要素**（element）と呼ばれています。

重要

配列は同じ型の値をまとめて記憶する機能をもつ。

配列

図9-1 配列
同じ型の値をまとめて記憶するには、配列を使います。

9.2 配列の宣言

配列を宣言する

それでは、配列を使ってみることにしましょう。変数と同じように、配列も利用する前に、

配列を宣言する

という作業が必要になります。

配列の宣言とは、配列の複数のハコを用意する作業にあたります。配列の場合も、識別子（第3章）から適当な配列名を用意し、型を指定します。

ただし、配列の場合はこれに加えて、

値をいくつ記憶できるのか

という指定も必要です。配列では、ハコの数を指定する必要があるのです。この数は、配列の**要素数**と呼ばれています。

配列の宣言は、次のようなスタイルになっています。

構文　配列の宣言

```
型名 配列名[要素数];
```

型名と要素数を指定します

それでは、int型の値を5つ記憶できる、要素数5の配列test[5]を宣言してみましょう。

```
int test[5];
```

int型の値を5つ記憶できる配列を宣言しました

配列の要素数は、決まった数でなければなりません。つまり、[ ]に記入する数は、5や10といった固定的な数である必要があります。

準備した配列のハコ（要素）には、それぞれのハコに次の名前がつけられます。

```
test[0]
test[1]
test[2]
test[3]
test[4]
```

[ ]の中に指定する番号は、**添字**（インデックス：index）と呼ばれています。この添字によって、配列のハコを特定することができるわけです。

C++の配列の添字は0からはじまるので、最後の添字は「要素数-1」となっています。つまり、5つの要素をもつ配列の場合は、

test[4]が値を格納できる最後の要素

になります。test[5]という名前の要素はありませんので、注意が必要です。

重要

配列は、型と要素数を指定して宣言する。
最後の配列要素の添字は、要素数より1つ小さい。

test[0] test[1] test[2] test[3] test[4]

図9-2 配列の宣言
5つの要素からなる配列を宣言すると、添字は0～4となります。

9.3 配列の利用

配列要素に値を代入する

それでは、準備した配列に値を格納してみましょう。配列の各要素はtest[0]、test[1]・・・という名前で扱えるようになっていますから、1つずつ値を代入してみます。この配列のハコには、整数値を代入することができます。

```
int test[5];
test[0] = 80;
test[1] = 60;
test[2] = 22;
test[3] = 50;
test[4] = 75;
```

配列を宣言しました

配列要素に1つずつ値を代入しています

ここでは、5つの配列要素にテストの得点を代入しています。配列要素に値を代入する方法は、変数と同じです。配列のハコを指定し、代入演算子の=を使って記述すればよいのです。

構文 配列要素への値の代入

```
配列名[添字] = 式;
```

重要

配列に値を記憶するには、添字を使って要素を指定し、値を代入する。

80
60
22
50
75
test[0]
test[1]
test[2]
test[3]
test[4]

図 9-3 配列への値の代入
配列に値を記憶させることができます。

配列要素の値を出力する

それでは、配列を実際に利用するコードを作成してみましょう。各要素の添字は0から順番に並んでいますので、第6章で学んだ繰り返し文を使うとすっきりと記述できます。繰り返し文を使って、配列に格納したテストの点数を出力するコードを作成してみることにします。

Sample1.cpp ▶ 配列要素の値を出力する

```
#include <iostream>
using namespace std;

int main()
{
   int test[5];

   test[0] = 80;
   test[1] = 60;
   test[2] = 22;
   test[3] = 50;
   test[4] = 75;

   for(int i=0; i<5; i++){
      cout << i+1 << "番目の人の点数は" << test[i] << "です。¥n";
   }

   return 0;
}
```

配列を宣言しています

配列要素に1つずつ値を代入します

繰り返し文を使って配列要素を出力しています

Sample1の実行画面

```
1番目の人の点数は80です。
2番目の人の点数は60です。
3番目の人の点数は22です。
4番目の人の点数は50です。
5番目の人の点数は75です。
```

配列要素の値が順番に出力されます

Sample1では、まず最初に配列の各要素に値を代入しています。そのあとで、for文を使って、各要素の値を出力します。配列の添字は0からはじまっているので、繰り返し文の中では出力する順番を「i+1番目」と指定しています。

このように配列では、

各要素を指定するときの添字に変数を使う

ことができるようになっています。これで、「何番目の学生が何点なのか」を繰り返し文の中で出力することができているわけです。配列と繰り返し文を使って、コードがすっきりと記述できていますね。

重要

配列と繰り返し文を使うと、多くのデータをかんたんに処理できる。

配列を初期化する

なお、配列にはいろいろな記述方法があります。Sample1では「配列の宣言」と「値の代入」を別々のコードに記述しました。配列では、この2つの作業ををまとめて行うことができるようになっています。これを**配列の初期化**といいます。

構文 **配列の初期化**

```
型名 配列名[要素数]={値0, 値1, ･･･ };
```

これまでみてきたテストの点数を扱う配列は、次のように初期化することができます。

```
int test[5] = {80,60,22,50,75};
```
5つの配列要素を初期化します

{ }の中に値をならべて各要素の値を記述します。すると、配列が準備されたときに80、60・・・という値が要素に格納されるのです。

{ }中に指定する値は、**初期化子**（initializer）と呼ばれています。上では80、60・・・が初期化子です。

また、配列を初期化する場合には、[ ]中に配列要素の数を指定しなくてもかまいません。要素数を指定しない場合は、

初期化子の数に応じた要素が自動的に準備される

ということになっているからです。つまり、次の初期化は上のコードとまったく同じ意味になります。

```
int test[] = {80,60,22,50,75};
```
要素数を指定しなくても5つの配列要素が用意されます

重要

配列を初期化すると、宣言と値の格納が同時に行える。

配列の添字に注意する

また、配列を使う場合には注意しておかなければならないことがあります。それは、

配列の大きさをこえた要素は利用することができない

ということです。たとえば、これまでのコードでは、5つの要素をもつ配列を宣言しました。この配列を扱うときは、test[10]などといった添字を指定して、値を代

入してはいけないのです。

```
int test[5];
//誤り
//test[10] = 50;
```

このような代入はできません

test[10]という要素は存在しません。このようなコードはまちがいとなります。配列の添字には注意するようにしてください。

重要

配列の大きさをこえる要素に値を代入しないようにする。

test[0] test[1] test[2] test[3] test[4]

?

test[10]

図9-4 配列要素への代入の注意
配列に値を代入する際には、添字の数値に注意する必要があります。

9.4 配列の応用

キーボードから入力する

今度は、テストの点数をキーボードから入力して配列に格納し、まとめて出力するコードを記述してみましょう。配列の数を変更しやすいように、定数を使ってみることにします。定数を使うには、変数の初期化を行うときにconstキーワードを指定するのでしたね。

Sample2.cpp ▶ 配列要素の数を入力する

```
#include <iostream>
using namespace std;

int main()
{
   const int num = 5;
   int test[num];

   cout << num << "人の点数を入力してください。¥n";
   for(int i=0; i<num; i++){
      cin >> test[i];
   }

   for(int j=0; j<num; j++){
      cout << j+1 << "番目の人の点数は" << test[j] << "です。¥n";
   }

   return 0;
}
```

人数の指定には定数を使うことにします

キーボードから５人の点数を入力します

入力した値を配列を使って出力します

Sample2 の実行画面

```
5人の点数を入力してください。
22 ↵
80 ↵
57 ↵
60 ↵
50 ↵
1番目の人の点数は22です。
2番目の人の点数は80です。
3番目の人の点数は57です。
4番目の人の点数は60です。
5番目の人の点数は50です。
```

このコードでは、キーボードから入力した値を配列に格納し、for文を使って出力しています。ここでは、まとめて点数を出力するのに配列の機能が役立っています。

このコードではconstを指定した定数を使っています。このため、テストを受けた人数にあわせてプログラムを書きかえるのはかんたんです。つまり、const int num = 5; の部分の「5」の数値を書きかえれば、さらに多くの人数の点数を管理するプログラムとすることができます。

配列の内容をソートする

もうひとつ、配列を使った応用をみてみましょう。テストの点数を並べかえてみることにします。値を順番に並べかえることを、**ソート**（sort）といいます。配列は要素に複数の値を格納することができるので、並べかえをするコードに利用すると便利なのです。

22 80 57 60 50

test[0] test[1] test[2] test[3] test[4]

Sample3.cpp ▶ 配列の要素をソートする

```
#include <iostream>
using namespace std;

int main()
{
   const int num = 5;
   int test[num];

   cout << num << "人の点数を入力してください。¥n";
    for(int i=0; i<num; i++){
      cin >> test[i];
   }

   for(int s=0; s<num-1; s++){
      for(int t=s+1; t<num; t++){
         if(test[t] > test[s]){
            int tmp = test[t];
            test[t] = test[s];
            test[s] = tmp;
         }
      }
   }

   for(int j=0; j<num; j++){
      cout << j+1 << "番目の人の点数は" << test[j] << "です。¥n";
   }

   return 0;
}
```

配列をソートしています

Sample3の実行画面

```
5人の点数を入力してください。
22 ↵
80 ↵
57 ↵
60 ↵
50 ↵
1番目の人の点数は80です。
2番目の人の点数は60です。
3番目の人の点数は57です。
4番目の人の点数は50です。
5番目の人の点数は22です。
```

このコードは、配列の要素を大きい順にソートするコードです。実行結果をみると、たしかに点数の高い順に出力されていますね。

配列をソートするにはさまざまな方法がありますが、ここでは次のような手法を使っています。順番にみてみることにしましょう。

①まず最初に、配列の各要素を配列の先頭の要素（test[0]）と比較していきます。比較した要素のほうが大きい場合は、先頭要素と入れかえます。すると、配列の先頭の要素に最大値を格納できます。

22	80	57	60	50
80	22	57	60	50
80	22	57	60	50
80	22	57	60	50

test[t]とtest[0]を比較して入れかえる。
test[t]のほうが大きい場合に入れかえる。
つまり、test[t] > test[s]（s=0）のとき入れかえる。

②これで、一番大きな値である配列の先頭要素が決まりました。そこで、残りの要素についても同じ処理を繰り返します。つまり、残りの要素を配列の2番目の要素（test[1]）と比較して、大きい場合に入れかえます。すると、最終的に2番目に大きい数値が2番目の要素となります。

80	22	57	60	50
80	57	22	60	50
80	60	22	57	50

test[t]とtest[1]を比較して入れかえる。
test[t]のほうが大きい場合に入れかえる。
つまり、test[t] > test[s]（s=1）のとき入れかえる。

③順に繰り返すと、配列のソートが完了します。

```
80 60 57 50 22
```

ちょっとややこしいですが、Sample3のコードと見くらべながら手順を確認してみましょう。Sample3では、繰り返し文をネストして、このソートの手順を記述しています。

要素を入れかえるためには、入れかえるものと同じ型の作業用のエリア（変数）が必要になります。そのため、このソートでは、作業用の変数tmpを使っています。

多次元配列のしくみを知る

ところでこれまでに学んだ配列は、一列に並んだハコのようなイメージでした。C++では、2次元以上にハコが並んだイメージをもつ**多次元配列**をつくることもできます。2次元配列であれば、表計算ソフトの表（ワークシート）のようなイメージを思いうかべるとよいでしょう。さらに、3次元配列であれば、X軸・Y軸・Z軸であらわされる立体を思いうかべるとよいかもしれません。

多次元配列の宣言は、次のようになります。

構文 **多次元配列（2次元の場合）**

```
型名 配列名[要素数][要素数];
```

実際の多次元配列の宣言をみてください。

```
int test[2][5];
```

この2次元の配列は、int型の値を2 × 5=10個記憶することができる配列です。多次元の配列は、さまざまな用途での利用が考えられるでしょう。たとえば、複数科目のテストの点を整理することができます。また、数学の行列式の計算などに応用することも考えられるでしょう。

ここではかんたんな例として、さきほどの5人分のテストの成績について、「国語」「算数」という2つの科目の点数を整理して格納してみましょう。2次元の配列に値を代入するようすをみてみます。

Sample4.cpp ▶ 多次元配列を使う

```
#include <iostream>
using namespace std;

int main()
{
   const int sub = 2;   // 科目数です
   const int num = 5;   // 人数です

   int test[sub][num];  // 科目数×人数分の値を格納する2次元配列を用意します

   test[0][0] = 80;
   test[0][1] = 60;
   test[0][2] = 22;
   test[0][3] = 50;
   test[0][4] = 75;
   test[1][0] = 90;     // 1つずつ値を代入します
   test[1][1] = 55;
   test[1][2] = 68;
   test[1][3] = 72;
   test[1][4] = 58;

   for(int i=0; i<num; i++){
      cout << i+1 << "番目の人の国語の点数は" << test[0][i] <<
         "です。¥n";
      cout << i+1 << "番目の人の算数の点数は" << test[1][i] <<
         "です。¥n";
   }
```

（注釈：test[0][i] → 国語の点数を出力します／test[1][i] → 算数の点数を出力します）

```
    return 0;
}
```

Sample4の実行画面

```
1番目の人の国語の点数は80です。
1番目の人の算数の点数は90です。
2番目の人の国語の点数は60です。
2番目の人の算数の点数は55です。
3番目の人の国語の点数は22です。
3番目の人の算数の点数は68です。
4番目の人の国語の点数は50です。
4番目の人の算数の点数は72です。
5番目の人の国語の点数は75です。
5番目の人の算数の点数は58です。
```

ここでは、test[0][●]に国語の点数を、test[1][●]に算数の点数を格納することにしました。これをネストしたfor文で繰り返して出力しています。2次元以上の配列への値の代入や出力も使いかたは基本的に同じです。

また、多次元配列を初期化することもできます。多次元配列では、通常の配列の初期化で使った{}の中に、もう一度{}を記述するかたちとなります。下のコードはSample5の多次元配列を初期化する記述です。

```
int test[2][5] = {
               {80,60,22,50,75},{90,55,68,72,58}
};
int test[][5] = {
               {80,60,22,50,75},{90,55,68,72,58}
};
```

省略することもできます

最初の配列の要素数については、1次元の配列と同じように、数を省略して記述することができるようになっています。多次元配列は、C言語の内部的には、1次元配列の各要素がさらに配列として扱われているので、このようなことができるのです。

Lesson 9

重要

多次元の配列を宣言して利用できる。

図9-5 多次元配列
配列は多次元にすることができます。

9.5 配列とポインタの関係

配列名のしくみを知る

ところで、第8章で学んだポインタと配列には密接な関係があります。この関係をみるために、配列の各要素のアドレスを調べることからはじめることにします。

第8章で学んだアドレス演算子（&）を思い出してください。変数と同じように、配列の各要素に対して&演算子を使うことができます。次のように、配列の要素に&演算子を使うと、その要素が格納されているアドレスを知ることができるのです。

```
&test[0]
&test[1]
```


上の表記で、配列の先頭要素（test[0]）、2番目の要素（test[1]）の値が格納されているアドレスを知ることができます。

さらに配列では、特別な書き方で配列要素のアドレスをあらわすことができるようになっています。「配列名」を記述しただけで、配列の先頭要素のアドレスをあらわすことができるのです。

```
test
```


[]をつけたり、添字をつけたりしてはいけません。&演算子も必要ありません。この表記のしくみを知るために、次のコードを入力してみましょう。

Sample5.cpp ▶ 配列名から先頭要素の値を知る

```
#include <iostream>
using namespace std;

int main()
{
   int test[5] = {80,60,55,22,75};

   cout << "test[0]の値は" << test[0] << "です。¥n";
   cout << "test[0]のアドレスは" << &test[0] << "です。¥n";
   cout << "testの値は" << test << "です。¥n";
   cout << "つまり*testの値は" << *test << "です。¥n";

   return 0;
}
```

配列の先頭要素のアドレスを&を使って知ることができます

つまり *test で配列の先頭要素をあらわせます

「配列名」でも先頭要素のアドレスをあらわします

Sample5 の実行画面

```
test[0]の値は80です。
test[0]のアドレスは0x00E4です。
testの値は0x00E4です。
つまり*testの値は80です。
```

&test[0]も test も先頭要素のアドレスをあらわします

配列の先頭要素をあらわします

出力結果をみてください。「test」の値が、「&test[0]」と同じであることがわかるでしょう。このように、「test」という配列名だけで、配列の先頭要素である「test[0]」のアドレスをあらわすことができるのです。

このように、配列名が先頭要素のアドレスをあらわしていることから、

配列名とは、配列の先頭要素のアドレスを格納しているポインタと同じはたらきをもつ

ということがいえます。配列名は、配列の先頭要素をさすポインタのように使えるのです。

さて、第8章でも学んだように、ポインタに間接参照演算子 (*) を使うと、それがさし示している変数の値を知ることができるのでした。配列名testに*をつけた場合にも、同じように配列の先頭要素であるtest[0]の値をあらわすことができるようになっています。実行結果で確認してみてください。

このことを次の表にまとめておきましょう。

test	&test[0]	→ 配列の先頭要素のアドレス
*test	test[0]	→ 配列の先頭要素の値

図9-6　配列名と先頭要素
配列名に * をつけると、配列の先頭要素の値をあらわすことができます。

ポインタ演算のしくみを知る

配列名は、配列の先頭要素へのポインタとして使えるということがわかりました。

ところで、C++では、ポインタが配列に密接な関係をもつとき、ポインタに対して次のような演算を行うことができるようになっています。表9-1をみてください。

表9-1：ポインタの演算

演算子	ポインタ演算の例（ただしp、p1、p2はポインタ）	
+	p+1	pがさしている要素の次の要素のアドレスを得る
-	p -1	pがさしている要素の前の要素のアドレスを得る
	p1-p2	p1とp2の間の要素数を得る

演算子	ポインタ演算の例（ただしp、p1、p2はポインタ）	
++	p++	pがさしている次の要素のアドレスを得る
--	p--	pがさしている前の要素のアドレスを得る

ポインタの演算は、第4章で学んだ四則演算などの計算とは少し異なります。たとえば+演算子をみてください。+演算子を使って「p+1」という計算をすると、

pがさしている「1つ次の要素」のアドレスを得る

という演算が行われるのです。「アドレスの値に1をたす」という計算をするのではありません。

では、次の例をみながら、ポインタの演算を行ってみることにしましょう。

Sample6.cpp ▶ ポインタの演算をする

```
#include <iostream>
using namespace std;

int main()
{
   int test[5] = {80,60,55,22,75};

   cout << "test[0]の値は" << test[0] << "です。¥n";
   cout << "test[0]のアドレスは" << &test[0] << "です。¥n";
   cout << "testの値は" << test << "です。¥n";
   cout << "test+1の値は" << test+1 << "です。¥n";
   cout << "*(test+1)の値は" << *(test+1) << "です。¥n";

   return 0;
}
```

先頭要素の「次の」要素のアドレスをあらわします

先頭要素の「次の」要素の値をあらわします

Sample6の実行画面

```
test[0]の値は80です。
test[0]のアドレスは0x00E4です。
testの値は0x00E4です。
test+1の値は0x00E8です。
*(test+1)の値は60です。
```

2番目の要素のアドレスとなっています

2番目の要素の値となっています

Sample6では、＋演算子を使ってポインタのたし算をしてみました。testに1をたした「test+1」を出力しています。すると、

1つ次の要素である、配列の2番目の要素のアドレス

が出力されています。さらに、＊演算子を使って*(test+1)を調べると、配列の2番目の要素の値を知ることができます。

つまり、*(test+1)は、配列の2番目の要素であるtest[1]と同じ意味になるわけです。

これらの知識を次の表にまとめておきましょう。下のように、配列は**添字を使った表記**と、**ポインタのたし算を使った表記**という2とおりの表記で、同じ要素をあらわすことができるようになっているのです。

*test	test[0]	→ 配列の先頭要素の値
*(test+1)	test[1]	→ 配列の2番目の要素の値
*(test+2)	test[2]	→ 配列の3番目の要素の値
・・・	・・・	・・・

図9-7 ポインタの演算
配列名（ポインタ）に＋演算子を使うと、たし算した数だけ先の要素をさし示すことができます。

ポインタと配列が密接な関係をもつとき、ポインタの演算を行うことができる。

配列名を使うときの注意

これまで説明したように、配列名は配列の先頭要素のアドレスを格納しているポインタと同じであると考えられます。ただし、このポインタは通常のポインタと異なる点があります。それは、

配列名であらわされるポインタには、ほかのアドレスを代入することができない

ということです。次のコードをみてください。

```
#include <iostream>
using namespace std;

int main()
{
   int a = 5;
   int test[5] = {80,60,55,22,75};

   //誤り
   //test = &a;

   return 0;
}
```


第8章で学んだように、通常のポインタには、ほかの変数のアドレスを代入することができました。しかし配列名は、その配列の先頭要素以外のアドレスをあらわすことができません。つまり、変数aのアドレスをtestに代入することはできないので注意してください。

9.6 引数と配列

配列を引数として使う

さて、配列とポインタには密接な関係があることがわかりました。この節では、この関係を利用したさまざまなコードをみていくことにしましょう。

まず、配列を関数の引数として使うコードをみることにします。次のコードを入力してみましょう。

Sample7.cpp ▶ 関数の引数として配列を使う

```
#include <iostream>
using namespace std;

//avg関数の宣言
double avg(int t[]);

int main()
{
   int test[5];

   cout << "5人のテストの点数を入力してください。¥n";
   for(int i=0; i<5; i++){
      cin >> test[i];
   }
   double ans = avg(test);
   cout << "5人の平均点は" << ans << "点です。¥n";

   return 0;
}

//avg関数の定義
double avg(int t[])
{
   double sum = 0;
```

配列を関数の引数として使います

配列名（配列の先頭要素のアドレス）を実引数として渡します

```
    for(int i=0; i<5; i++){
        sum += t[i];    ← 配列を利用します
    }
    return sum/5;
}
```

Sample7の実行画面

```
5人のテストの点数を入力してください。
80 ↵
60 ↵
55 ↵
22 ↵
75 ↵
5人の平均点は58.4点です。
```

この5人のテストの平均点を返すavg()関数では、配列を引数として使っています。このとき、仮引数にはt[]と記述し、実引数として配列名testを記述して渡していることに注目してください。このように、引数として配列を使うときには、実引数として配列名を渡します。

さきほど学んだように、配列名は先頭要素のアドレスをあらわすので、関数には配列の名要素の値ではなく、配列の先頭要素のアドレスだけが渡されることになります。配列を関数に渡す場合は、この先頭要素のアドレスを渡す方法を使うので注意してください。

ポインタを引数として使う

前の節でみた配列とポインタの密接な関係を使うと、同じ関数をポインタのように記述することができます。次のコードをみてください。

```
//avg関数の宣言
double avg(int* pT);
...
```

```
//avg関数の定義
double avg(int* pT)
{
   double sum = 0;
   for(int i=0; i<5; i++){
      sum += *(pT+i);
   }
   return sum/5;
}
```

仮引数をポインタとして表記することもできます

配列をポインタの演算による表記で利用することができます

この関数の仮引数は、ポインタpTとしています。このとき、呼び出し元から配列の先頭要素のアドレスが渡されると、ポインタpTがそのアドレスで初期化されます。このため関数内では、ポインタpTに対してポインタ演算を行うことで、配列要素を処理できるようになります。

つまり、このavg()関数は、Sample7のavg()関数のかわりとして使うことができるのです。全体のコードは次のようになります。

```
#include <iostream>
using namespace std;

//avg関数の宣言
double avg(int* pT);

int main()
{
   int test[5];
   cout << "5人のテストの点数を入力してください。¥n";
   for(int i=0; i<5; i++){
      cin >> test[i];
   }

   double ans = avg(test);
   cout << "5人の平均点は" << ans << "点です。¥n";

   return 0;
}

//avg関数の定義
double avg(int* pT)
{
   double sum = 0;
   for(int i=0; i<5; i++){
```

仮引数をポインタで表記して・・・

```
        sum += *(pT+i);
    }
    return sum/5;
}
```

ポインタの演算によって扱います

avg()関数を利用している部分のコードは、Sample7とまったく同じです。配列を受けとる関数は、このようにポインタで表記した仮引数を用いることがよくあります。

```
int main()
{
   double ans = avg(test);
}
double avg(int* pT)
{
  ...
}
```

0x00E4

0x00E4

pT

図9-8 引数と配列
仮引数としてポインタを使い、配列の先頭要素のアドレスを受けとることができます。

ポインタに[]演算子を使う

これまでのところでは、配列名を先頭要素へのポインタとして扱うことができることをみてきました。実は、これとは逆に、

ポインタに[]を使って配列のように表記する

ということもできます。たとえば、前のavg()関数の中のポインタpTをみてください。このポインタの表記に[]を使って、次のような表記ができます。

```
pT[2]
```

ポインタに[]を使うことができます

この[]を**添字演算子**（subscript operator）ともいいます。このようにポインタに[]を使った表記は、

ポインタpTがさしている要素から数えて、2つ先の要素

をあらわすことになっています。ポインタが配列をさしているときには、ポインタに[]をつけて、それがさし示す要素をあらわすことができるようになっているのです。ちょうど配列と同じ表記になるわけです。

ただし、[]を正しく使えるのは、

ポインタと配列が密接な関係をもつときだけ

なので注意してください。この関数では、実引数として配列の先頭要素のアドレスが渡されてくるので、[]を使う表記ができるのです。ほかのポインタに[]を使うと、プログラムを実行したときに思いがけないエラーがおこってしまう場合があるので、注意して使うようにしてください。

下のコードはさきほど作成したavg()関数内のポインタに、[]演算子を適用して、配列のようにあらわしたものです。


```
//avg関数の定義
double avg(int* pT)
{
   double sum = 0;
   for(int i=0; i<5; i++){
      sum += pT[i];
   }
   return sum/5;
}
```

ポインタに[]を使った表記ができます

繰り返し文の中で、pT[i]という表記を使うことによって、ポインタpTがさす配列の要素に順番にアクセスしています。つまり、この関数は前のavg()関数とまったく同じ処理を行っています。

このように、ポインタと配列に関係がある場合は、ポインタと配列を同じように扱うことができるのです。

9.7 文字列と配列

文字列と配列の関係を知る

この章の最後では、配列を応用したしくみについて学ぶことにしましょう。

実は、いままで何度も登場したC++の「文字列」は、配列と深いかかわりをもっています。文字列とは、

```
"Hello"
```

のような文字の並びのことでしたね。C++では、このような英数字の文字列を、

「char型の配列」で扱うことができる

というしくみになっています。「char型」は、第3章でも紹介したように、「文字」をあらわす型です。文字列は文字の並びからなりたっているので、この型の配列で扱うことができるのです。

たとえば、上の"Hello"という文字列は、次のようにして扱います。

```
char str[6];        ← char型の配列を宣言します

str[0]='H';
str[1]='e';
str[2]='l';         ← 1つずつ文字を代入します
str[3]='l';
str[4]='o';
str[5]='¥0';        ← 最後に'¥0'をつけます
```

char型の配列を宣言したあと、各要素に1つずつ文字を代入しています。これで、配列str[]で"Hello"という文字列を扱えるようになるのです。

ただし、C++では日本語の文字列は扱いが異なる場合があるので、ここでは英数字だけを扱うことにしておきます。

ところで、この配列の最後に代入した'¥0'とは何でしょうか？この文字は**NULL文字**（NULL character）と呼ばれています。C++では文字列配列の最後をあらわすために、

¥0

必ず最後に'¥0'という値をつける

ことになっているのです。

この「'¥0'」を忘れないでください。特に配列を宣言するときには、注意が必要です。この「'¥0'」の分のハコも忘れずに用意する必要があるからです。

たとえば、"Hello"のような5文字からなる文字列を扱う場合を考えてみてください。この場合、5文字からなる文字列だからといって、要素を5個しかもたない配列では文字列を扱うことができません。このとき、配列要素は最低6個用意する必要があります。最低「**文字列の長さ+1**」個の要素が必要となるのです。

重要
文字列はchar型の配列として扱う。
文字列配列の最後の要素は'¥0'とする。
文字列配列の要素数は、「文字列の長さ+1」以上必要となる。

図9-9 文字列配列
文字列配列の最後の要素はNULL文字（¥0）とします。

文字列配列を初期化する

文字列は次のように初期化して、char型の配列に格納することもできます。

```
char str[6] = {'H','e','l','l','o','¥0'};
char str[] = {'H','e','l','l','o','¥0'};
char str[6] = "Hello";
char str[] = "Hello";
```

配列の初期化をすることができます

""を使って初期化することもできます

これらは、どれも"Hello"という文字列を扱うために、char型の配列str[6]を初期化したものです。文字列の場合は、通常の配列の初期化方法のほか、上のように

" "でかこんで初期化する

という方法を使うことができるようになっているのです。" "を使って文字列配列を初期化した場合は、自動的にNULL文字（¥0）が付け加えられることになっています。

なお、" "を使う格納方法は、初期化のときにだけ利用することができます。次のように、" "を使ってあとから文字列を代入することはできませんので、注意してください。

```
char str[6] = "Hello";
/* 誤り */
/* str = "Hello" */
```

" "を使って初期化ができますが・・・

" "を使って代入することはできません

文字列配列を出力する

それでは、配列に格納した文字列を出力してみることにしましょう。まず、次のようなコードを入力してください。

Sample8.cpp ▶ 文字列を出力する

```
#include <iostream>
using namespace std;

int main()
{
   char str[] = "Hello";

   cout << str <<'¥n';     // 文字列を出力します

   return 0;
}
```

Sample8の実行画面

```
Hello
```

coutに<<記号を使って上のように記述すると、文字列全体が出力されました。

文字列をポインタで扱う

文字列は、配列のかわりにchar型へのポインタを使って扱うこともできます。つまり、次のコードのように文字列を扱うことができるのです。

```
char* str = "Hello";
```

この場合のstrは、char型へのポインタをあらわしていることに注意してください。char型へのポインタに代入する場合は、次のようなイメージになっています。ポインタで扱う場合には、" "を使うことでメモリ上のどこか別の場所に文字列が格納され、その場所をさし示すようになっています。

図9-10 文字列の扱い
char str[]="Hello"（配列）で文字列を扱うことができます（上）。
char* str="Hello"（ポインタ）で文字列を扱うことができます（下）。

文字列をchar型へのポインタとして扱った場合にも、文字列を出力することができます。ためしてみましょう。

Sample9.cpp ▶ ポインタによる文字列の出力

```
#include <iostream>
using namespace std;

int main()
{
   char* str = "Hello";

   cout << str <<'¥n';     // 文字列を出力します

   return 0;
}
```

出力結果は、Sample8と同じになります。

配列とポインタの違いを知る

文字列を配列で扱う場合と、ポインタで扱う場合には、いくつか注意をしておかなければならないことがあります。

ポインタに文字列（の先頭要素）を" "を使って格納した場合は、配列に格納した場合と異なり、さらに" "を使って扱う文字列を変更することができるようになっています。

```
char* str = "Hello";
str = "Goodbye"
```

" "を使ってポインタを文字列で初期化することができます

" "を使ってポインタがさす文字列を変更できます

このコードでは、1行目でstrが "Hello"をさすようにしています。2行目では、strは"Goodbye"をさすようにしています。

次の図をみてください。ポインタの場合は、どこか別の場所に格納された文字列をさしていますから、ポインタのさし示す先を変更するだけで扱う文字列を変更できるようになっているのです。

配列名に代入はできない

ポインタに代入すると
ポインタの示す先を変更できる

図9-11 配列とポインタの違い
文字列を配列で扱う場合には、配列名に代入をして文字列を変更することはできません（上）。
文字列をポインタで扱う場合には、ポインタに代入してさし示す文字列を変更することができます（下）。

また、ここでは出力方法について説明しましたが、文字列を入力する場合には、配列を宣言して、文字列を格納する領域をきちんと確保しておかなければなりません。

```
char str[100];
...
cin >> str;
```


文字列は配列とポインタの2とおりで扱うことができるのですが、しくみの違いに注意しながら、コードを作成していく必要があります。

文字列を操作する

では今度は、文字列が配列であることを利用しながら、文字列を操作するコードを作成してみることにしましょう。次のコードは、配列str[]に記憶した文字列"Hello"の各文字の間に*という記号を挿入して、出力します。

Sample10.cpp ▶ 配列で文字列を扱う

```
#include <iostream>
using namespace std;

int main()
{
   char str[] = "Hello";

   cout << "Hello¥n";

   for(int i=0; str[i]!='¥0'; i++){
      cout << str[i] << '*';
   }
   cout << '¥n';

   return 0;
}
```

¥0でない限り繰り返します

Sample10の実行画面

```
Hello
H*e*l*l*o*
```

このコードでは、for文を使って文字列配列の要素を1個ずつ扱っています。C++の文字列は必ず¥0で終わるため、配列str[]の要素に¥0が登場するまで＊を挿入しながら、配列要素を繰り返し出力していくことができます。

¥0でない限り繰り返します

```
for(int i=0; str[i]!='¥0'; i++){
   cout << str[i] << '*';
```

```
}
```

このように文字列を扱うコードの場合は、最後の文字が¥0であることを利用します。ほかの記号を挿入したりして、文字列の扱いに慣れておくとよいでしょう。

重要

文字列を操作するときは、最後が ¥0 であることを利用する。

'H' 'e' 'l' 'l' 'o' '¥0'
str[0] str[1] str[2] str[3] str[4] str[5]

図9-12 文字列の操作
C++の文字列は最後が¥0であることを利用して操作します。

文字列を扱う標準ライブラリ関数

ここまで、文字列の操作方法をいくつかみてきました。お使いのC++の環境には、文字列を扱うための標準的な関数が添付されています。

この関数群は**標準ライブラリ**（standard library）と呼ばれています。標準ライブラリには、さまざまな文字列操作関数が定義されています。これをコード中で利用すると、文字列の長さを調べる作業やコピーを行う作業がかんたんに行えるようになっています。標準ライブラリの主な文字列操作関数を紹介しておきましょう。

表9-2：文字列を扱う標準ライブラリ関数（<string>）

size_t strlen(const char* s);
文字列sのNULL文字を除く長さを返します。
char* strcpy(char* s1, const char* s2);
文字列s2を配列s1にコピーしてs1を返します。
char* strcat(char* s1, const char* s2);
文字列s2を文字列s1の末尾に追加してs1を返します。
int strcmp(const char* s1,const char* s2);
文字列s1と文字列s2を比較し、 文字列s1が文字列s2より小さいとき　：負の値 文字列s1が文字列s2と等しいとき　：0 文字列s1が文字列s2より大きいとき　：正の値 を返します。

標準ライブラリ関数を利用するためには、各関数の機能が提供されているファイルをインクルードします。文字列操作関数を利用するためには、#include <string>と記述してください。

それでは、このうちの**strlen()関数**を利用して「文字列の長さを求める」コードを入力してみましょう。

Lesson 9

Sample11.cpp ▶ 標準ライブラリのstrlen()関数を使う

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   char str[100];

   cout << "文字列（英数字）を入力してください。¥n";

   cin >> str;

   cout << "文字列の長さは" << strlen(str) << "です。¥n";

   return 0;
}
```

文字列操作関数を提供する標準ライブラリを扱えるようにします

文字列操作関数を利用します

Sample11 の実行画面

```
文字列（英数字）を入力してください。
Hello ↵
文字列の長さは5です。
```

このように標準ライブラリstrlen()関数を使うと、文字列の長さを調べることができます。strlen()関数で返される文字列の長さとは、'¥0'を含まない、文字だけの個数であることに注意してください。

文字列を配列にコピーする

次に、文字列を配列にコピーする関数を使ってみましょう。さきほど紹介したように、配列を初期化する場合には、" "を使って文字列を配列に格納することができます。しかし初期化以外の場合では、文字列を配列に格納するのは面倒な作業になります。そこで、文字列を配列にコピーする標準ライブラリ関数を使うと便利です。次のコードをみてみましょう。

Sample12.cpp ▶ 標準ライブラリのstrcpy()関数を使う

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   char str0[20];
   char str1[10];
   char str2[10];

   strcpy(str1, "Hello");      ● str1[]に“Hello”をコピーします
   strcpy(str2, "Goodbye");    ● str2[]に“Goodbye”をコピーします
   strcpy(str0, str1);         ● str0[]にstr1[]をコピーします
   strcat(str0, str2);         ● str0[]の末尾にstr2[]を追加します

   cout << "配列str1は" << str1 << "です。¥n";
```

```
    cout << "配列str2は" << str2 << "です。¥n";

    cout << "連結すると" << str0 << "です。¥n";

    return 0;
}
```

Sample12の実行画面

```
配列str1はHelloです。
配列str2はGoodbyeです。
連結するとHelloGoodbyeです。
```

ここでは、文字列を配列にコピーする**strcpy()関数**を使いました。さらに、配列の末尾に文字列を追加する**strcat()関数**も使っています。ここでは、str1[]に"Hello"を、str2[]に"Goodbye"をコピー（格納）しました。

さらに、str0[]にstr1[]をコピーして、str0[]に"Hello"が格納されるようにしました。最後に、str0[]の末尾に"Goodbye"が追加されるようにstrcat()関数を使ったのです。この手順を図で示すと次のようになります。

strcpy(str1,"Hello");

str1	H	e	l	l	o	¥0							

strcpy(str2,"Goodbye");

str2	G	o	o	d	b	y	e	¥0		

strcpy(str0,str1);

str0	H	e	l	l	o	¥0							

strcat(str0,str2);

str0	H	e	l	l	o	G	o	o	d	b	y	e	¥0		

文字列はこのような標準ライブラリ関数によって扱うと便利です。これらの関数を使えば、文字列をコピーしたり、配列に格納する操作をシンプルに行うことができます。

配列の大きさに注意する

strcpy()関数やstrcat()関数を使う場合には、配列の大きさに十分気をつけてください。関数の引数がさす配列の大きさが十分でないと、これらの関数は配列の要素をこえて文字列をコピーしてしまいます。たとえば、Sample3の場合にstr0[]の要素が10個だったとしましょう。

```
int main()
{
   char str0[10];
   char str1[10];
   char str2[10];
   ...
```

十分でない大きさの配列を使ってはいけません

この場合、最後にstrcat()関数を使って"Goodbye"をコピーしたときに、配列の最大要素をこえて文字列を格納する操作を行ってしまうことになります。この章の最初でも述べたように、配列の要素をこえて要素を扱うような操作を行ってはいけません。このように記述したコードは問題なくコンパイルできてしまうのですが、プログラムを実行した際に思いもかけないエラーがおこる場合があり、たいへん危険です。

str2 | G | o | o | d | b | y | e | ¥0 |

strcat(str0,str2);

str0 | H | e | l | l | o | G | o | o | d | b | y | e | ¥0 |

この部分の関係のないメモリ領域のデータが破壊されてしまいます

特に、文字列の末尾に追加するstrcat()関数などでは、誤って配列の大きさをこえてしまいがちです。大きさに余裕のある配列に追加するように注意してください。

重要

文字列を配列にコピーする場合は、配列の大きさをこえないように気をつける。

9.8 レッスンのまとめ

この章では、次のようなことを学びました。

- 配列を宣言して、各要素に値を代入できます。
- { }内に初期化子を指定して、配列を初期化することができます。
- 多次元配列を作成して扱うことができます。
- 配列とポインタには密接な関係があります。
- 配列名は、配列の先頭要素のアドレスをあらわします。
- 配列を引数にもつ関数には、配列の先頭要素のアドレスを渡します。
- 文字列配列の末尾には、'¥0'を格納します。
- 文字列配列は、" "を使った文字列を指定して初期化できます
- 標準ライブラリの文字列操作関数を利用することができます。

配列を利用すると、同じ種類の多くのデータを扱うことができます。また、C++では、文字列を利用するときにも、配列の機能が欠かせません。さらに、この章では配列とポインタとのかかわりについても学びました。配列は、C++に欠かせない便利なしくみなのです。

練習

1. 5つの要素をもつ配列を受けとり、その最大値を返すint max(int x[])関数を定義してください。max()関数を使い、キーボードから学生の数とテストの点数を入力させ、最高点を出力するコードを記述してください。

```
5人のテストの点数を入力してください。
50 ↵
20 ↵
35 ↵
68 ↵
75 ↵
最高点は75点です。
```

2. 文字列の長さを調べる関数int length(char* str)を作成してください。実際にキーボードから文字列を入力して文字列の長さを調べるコードを記述してください。

```
文字列を入力してください。
Hello ↵
文字列の長さは5です。
```

3. 文字列に指定した文字がいくつあるかを調べる関数int count(char str[], char c)を作成してください。実際にキーボードから文字列を入力して文字の個数を調べるコードを記述してください。

```
文字列を入力してください。
Hello ↵
文字列から探す文字を入力してください。
l ↵
Helloの中にlは全部で2個あります。
```

Lesson 9

Lesson 10

大きなプログラムの作成

これまでの章では、小さなプログラムを数多く作成してきました。しかし、プログラムが大規模なものになると、コード中でたくさんの変数・配列・関数を扱っていくことになるでしょう。この章では、大きなプログラムを作成するために必要な知識を学ぶことにします。

Check Point!

- スコープ
- 記憶寿命
- 動的なメモリの確保
- new演算子
- delete演算子
- 分割コンパイル
- ヘッダ
- リンケージ

10.1 変数とスコープ

変数の種類を知る

さてここでは、関数の中で変数や配列を宣言するしくみについて学んでおくことにしましょう。変数・配列の宣言は、これまでmain()関数や自分で作成した関数の中に記述してきたことを思い出してください。

```
int main()
{
    int a;  ● 関数ブロック内に宣言したローカル変数です
    ...
}
```

ただし、これは必ず関数の中で行わなければならないわけではありません。変数や配列は、

関数の外側で宣言をする

ということもできるようになっています。

```
int a;  ● 関数ブロックの外で宣言したグローバル変数です

int main()
{
    ...
}
```

これまでのように、関数の中で宣言した変数を**ローカル変数**（local variable）と呼びます。一方、関数の外で宣言した変数を**グローバル変数**（global variable）と呼んでいます。

重要

関数の中でローカル変数を宣言できる。
関数の外でグローバル変数を宣言できる。

```
int a = 0;            ―― グローバル変数

int main()
{
   int b = 1;         ―― ローカル変数
    ...
}

void func()
{
   int c = 2;         ―― ローカル変数
    ...
}
```

図10-1　ローカル変数とグローバル変数
関数の中で宣言した変数を「ローカル変数」と呼びます。関数の外で宣言した変数を「グローバル変数」と呼びます。

スコープのしくみを知る

では、ローカル変数とグローバル変数にはどのような違いがあるのでしょうか？ 変数の違いをためしてみるために、2種類の変数を扱うコードを記述してみましょう。次のコードをみてください。

Sample1.cpp ▶ 変数のスコープを知る

```
#include <iostream>
using namespace std;

//func関数の宣言
void func();

int a = 0;        ← グローバル変数aです

//main関数
int main()
{
```

```
    int b = 1;

    cout << "main関数では変数aとbが使えます。¥n";
    cout << "変数aの値は" << a << "です。¥n";
    cout << "変数bの値は" << b << "です。¥n";
    //cout << "変数cの値は" << c << "です。¥n";

    func();

    return 0;
}

//func関数の定義
void func()
{
    int c = 2;

    cout << "func関数では変数aとcが使えます。¥n";
    cout << "変数aの値は" << a << "です。¥n";
    //cout << "変数bの値は" << b << "です。¥n";
    cout << "変数cの値は" << c << "です。¥n";
}
```


Sample1 の実行画面

```
main関数では変数aとbが使えます。
変数aの値は0です。
変数bの値は1です。
func関数では変数aとcが使えます。
変数aの値は0です。
変数cの値は2です。
```

このコードでは、次の3つの変数を宣言しています。

変数a ・・・ 関数の外に宣言したグローバル変数

変数b ・・・ main()関数内に宣言したローカル変数

変数c ・・・ func()関数内に宣言したローカル変数

変数は宣言した位置によってコード内のどこで利用できるのかが異なります。

まず、ローカル変数の場合は、

宣言した関数内だけで利用することができる

ということになっています。たとえば、ローカル変数bはmain()関数内で宣言されていますから、func()関数内では利用することができません。逆にローカル変数cはmain()関数内では利用することができません。

一方、グローバル変数は、

どの関数からでも利用することができる

ということになっています。つまり、変数aはmain()、func()関数のどちらからでも利用することができるのです。

図10-1は、それぞれの変数が利用できる部分の範囲を示しています。このように変数の名前が通用する範囲のことを**スコープ**（scope）と呼んでいます。

```
int a = 0;

int main()
{
    int b = 1;

    func();

    return 0;
}

void func()
{
    int c = 2;
}
```

グローバル変数aのスコープ
ローカル変数bのスコープ
ローカル変数cのスコープ

図10-1　変数のスコープ
ローカル変数は変数を宣言した関数内だけで利用できます。グローバル変数はどの関数からでも利用できます。

ローカル変数とグローバル変数のスコープについてまとめておきましょう。

表10-1：スコープ

	宣言位置	スコープ
ローカル変数	関数の中	変数を宣言した場所から関数の終わりまで利用できる
グローバル変数	関数の外	どの関数からでも利用できる

重要

変数の名前が通用する範囲をスコープという。

ローカル変数の名前が重なると？

変数の名前と宣言位置には注意するようにしてください。たとえば、同じ関数内のローカル変数には同じ名前をつけることはできません。ただし、異なる関数内で宣言したローカル変数には同じ名前をつけてもかまいません。

```
void main()
{
   int a = 0;
   a++;
}
int func()
{
   int a = 0;
   a++;
}
```

2つのローカル変数は、まったく別のものです

上のコードでは、main()関数とfunc()関数でともに「変数a」を宣言しています。この2つのローカル変数は、異なる値を格納することができる、まったく別の変数をあらわしています。同じ名前がついていても、異なる関数内のローカル変数は、別の変数なのです。

重要

異なる関数内のローカル変数は、同じ名前であっても異なる変数を意味する。

```
int main()
{
   int a = 0;
   a++;
}
```

```
void func()
{
   int a = 0;
   a++;
}
```

図10-2 ローカル変数の名前の重複
異なる関数内で宣言されたローカル変数は、まったく異なる２つの変数となっています。

グローバル変数と名前が重なると？

グローバル変数とローカル変数は、同じ変数名を使ってもかまいません。次のコードをみてください。

```
int a = 0;          ← グローバル変数aです

//main関数
int main()
{
   int a = 0;       ← ローカル変数aです
   ...
   a++;             ← インクリメントされるのはローカル変数aです

   ::a++;           ← インクリメントされるのはグローバル変数aです
}
```

ここではグローバル変数として「変数a」を宣言し、main()関数内でさらにローカル変数として「変数a」を宣言しました。グローバル変数とローカル変数では、このように名前を重複させることができます。

ただし、main()関数内で「a++;」などという記述をすると、それはローカル変数

aのことをさします。ちょうどfunc()関数内では、ローカル変数によってグローバル変数の名前が隠されるようになっていることに気をつけてください。グローバル変数を使うには「::a++;」というように、グローバル解決演算子（::）を使います。

```
int a = 0;

int main()
{
    int a = 0;
    ...
    a++;

    ::a++;
}
```

「a」はグローバル変数aを意味する

「a」はローカル変数aを意味する
「::a」はグローバル変数aを意味する

図 10-3 グローバル変数とローカル変数の名前の重複
グローバル変数と関数内のローカル変数の名前が重複すると、その関数内でグローバル変数の名前が隠されます。

重要

グローバル変数とローカル変数で同じ名前を使うことができる。

10.2 記憶寿命

変数の記憶寿命を知る

ところで、変数や配列は、プログラムを開始してから終了するまで、ずっと続けて値を記憶しているわけではありません。ここで変数の「一生」をみてみることにしましょう。

変数を宣言すると、まず値を記憶するためのハコがメモリ内に準備されます(①)。これを**メモリが確保される**ということもあります。そのあと、変数に値を格納したり出力したりして利用するわけですが(②)、最後にハコが破棄されることによって、メモリがまた別の用途に使われるようになります(③)。これを**メモリが解放される**ともいいます。

変数のハコが存在し、値を記憶していられる期間のことを**記憶寿命**といいます。

図10-4 変数の記憶寿命
①値を記憶するためのハコがメモリ内に準備され、②変数に値を格納したり出力したりして利用します。③最後に変数のハコは破棄されて、メモリは別の用途に使われます。

変数がどのような記憶寿命をもつのかということは、変数の宣言位置にも関係しています。

通常のローカル変数は、

関数内で宣言されたときに、変数のハコがメモリ内に準備され

↓

関数が終了する際にハコが破棄されてメモリが別の用途に使われるようになる

という一生をたどります。つまり、通常のローカル変数は、宣言されてから関数が終了するまでの間だけ、値を格納しておくことができるのです。

ローカル変数b

関数のはじまり　変数の宣言　関数の終了　関数のはじまり　変数の宣言　関数の終了

b　b　b　b

一方グローバル変数は、

プログラムの本体の処理がはじまる前に、一度だけメモリが確保され

↓

プログラムの終了時にメモリが解放される

という一生をたどります。つまり、グローバル変数はプログラムの開始から終了するまでの間、ずっと値を格納しておくことができるのです。

変数の一生をたしかめるために、次のコードをみてみましょう。

Sample2.cpp ▶ 変数の記憶寿命を知る

```
#include <iostream>
using namespace std;

//func関数の宣言
void func();

int a = 0;   ● グローバル変数aです

//main関数
int main()
{
   for(int i=0; i<5; i++)
      func();

   return 0;
}

//func関数の定義
void func()
{
   int b = 0;   ● ローカル変数bです
   static int c = 0;   ● staticをつけたローカル変数cです

   cout << "変数aは" << a << "変数bは" << b << "変数cは" << c
      << "です。¥n";
   a++;
   b++;   各変数をインクリメントしています
   c++;
}
```

Sample2の実行画面

```
変数aは0変数bは0変数cは0です。
変数aは1変数bは0変数cは1です。
変数aは2変数bは0変数cは2です。
変数aは3変数bは0変数cは3です。
変数aは4変数bは0変数cは4です。
```

ローカル変数bはインクリメントされません

func()関数は変数a、b、cの値を出力し、1つずつインクリメントしている関数です。グローバル変数aはプログラムの開始から終了まで値を記憶しているので、1つずつ値が増えていきます。

一方ローカル変数bは、関数が呼び出されるたびに最初に0が格納され、関数の終了ごとにハコが破棄されます。このため、インクリメントをしても、いつも0のままになっています。

staticをつけると？

このように通常のローカル変数は関数が終了するまでの記憶寿命しかもちません。ただし、ローカル変数に**static**というキーワードを指定をすると、グローバル変数と同じ記憶寿命をもつようにすることができます。このようなローカル変数を、**静的寿命をもつローカル変数**といいます。

Sample2のコードの変数cは静的寿命をもつローカル変数です。この変数は、グローバル変数と同じようにプログラムの開始時に初期化され、終了時に破棄されることになっています。つまり、func()関数が終了してもハコが破棄されず、値を保持したままなので、関数が呼び出されるたびに1つずつ数が増えていきます。staticは**記憶クラス指定子**（storage class identifier）と呼ばれています。

それではここで、変数の記憶寿命に関することがらをまとめておきましょう。

表10-2：記憶寿命

	記憶クラス	記憶寿命
ローカル変数	（自動）	宣言されてから関数が終了するまで（自動）
	static	プログラムの実行準備時から終了まで（静的）
グローバル変数		プログラムの実行準備時から終了まで（静的）

図10-5 変数の寿命
グローバル変数、静的なローカル変数はプログラムの開始から終了までの寿命をもちますが、通常のローカル変数は関数内の寿命をもちます。

なお、変数の初期化のコードを記述しなかった場合には、グローバル変数とstaticなローカル変数は、自動的に0で初期化されることになっています。一方、ローカル変数を初期化しない場合の初期値は、特に決まっていません。

ローカル変数のバリエーション

ローカル変数は関数の先頭だけでなく、for文やif文などのブロックの先頭でも宣言することができます。関数の仮引数もローカル変数の一種です。

ローカル変数は、それを宣言したブロック内（{ }）だけで通用する変数となります。ブロックの内側と外側で変数名が重なったときには、内側の変数の名前が優先されます。

```
void func(int x)
{
  int a;

  for(int i = 0; i<10; i++){
    int s;
    ...

  }
}
```

ローカル変数へのポインタと戻り値

ところで、第7章で学んだ関数では、ポインタや参照を使うことができます。しかしこのときには注意が必要です。次のコードをみてください。

```
//func関数の定義
int* func()
{
    int a = 10;
    int* pA = &a;
    return pA;
}

int main()
{
    int* pRes = func();
    ...
}
```

ポインタpAは通常のローカル変数aをさしていますので・・・

pResには意味のないアドレスが格納されてしまいます。このような利用はできません

ここではfunc()関数の戻り値を、func内で宣言したローカル変数のaをさすポインタ（pA）としています。

しかし、func()関数が終了したときに通常のローカル変数であるaは破棄されてしまいます。このため、呼び出し元のところに戻ったときには変数aのアドレスは意味のない値となってしまうのです。つまり、pResは意味のないポインタとなってしまうのです。

このようなpResを利用したコードを記述すると、思いもかけないエラーが起こる場合があります。戻り値には気をつける必要があるのです。

一方、次のような戻り値を利用することは可能です。

```
//func関数の定義
int* func()
{
    static int a = 10;
    int* pA = &a;
    return pA;
}

int main()
{
    int* pRes = func();
    ...
}
```

このポインタは静的なローカル変数aをさしていますので・・・

このような利用が可能です

このコードでも、関数内で宣言したローカル変数へのポインタを戻しています。しかし、この変数aはstaticをつけた静的な変数となっています。この変数は関数が終了しても存在しているため、この変数へのポインタを呼び出し元でも利用することができるわけです。

同じ理由で、通常のローカル変数への参照を戻すこともできません。参照は変数の別名にすぎないからです。変数が破棄されていれば、参照も意味のないものとなってしまいます。

重要

通常のローカル変数へのポインタや参照を戻り値としてはいけない。

10.3 動的なメモリの確保

動的なメモリの確保

10.2節でみたように、グローバル変数はプログラムの実行開始時にメモリが確保されます。また、通常のローカル変数は関数が呼び出されて宣言されたときにメモリが確保されます。

実は、これらの2とおりの方法以外にも、私たちプログラムを作成する人間が指定するタイミングで、メモリを確保する方法があります。これを**動的なメモリの確保**（dynamic allocation）といいます。

図10-6 **new演算子によるメモリの確保**
new演算子を使うと、指定した時点でメモリを確保できます。

この方法を使うと、これまで説明したような変数を利用したメモリの利用と異なり、自分でメモリを確保するタイミングを決定することができます。

動的にメモリを確保するには、newという演算子を使います。

構文 **new演算子によるメモリの確保**

```
ポインタ = new 型名;
```

動的にメモリを確保するコードをみてみましょう。

```
int* pA;
pA = new int;
```


動的にメモリを確保するには、まず、確保する型へのポインタ（ここではpA）を用意しておく必要があります。

new演算子はメモリを確保し、その確保したメモリのアドレスを返します。そこで、このアドレスをポインタpAに格納しているのです。

なお、動的にメモリを確保した場合は変数を利用した場合と異なり、このアドレスを使ってメモリを直接扱う必要があります。つまり、確保したメモリに何か値を記憶させるときは、そのポインタを使って、次のように代入します。

```
*pA = 50;
```

このようにポインタを介して値を代入する方法は、すでに第8章で学びましたね。

動的なメモリの解放

動的にメモリを確保した場合には、注意しなければならないことがあります。メモリが必要でなくなった場合、私たちプログラムを作成する人間がこのメモリを解放する処理を記述しなければなりません。ローカル変数やグローバル変数と異なり、メモリをいつ解放するかを私たちが決めてやらなければならないのです。

図10-7 delete演算子によるメモリの解放
delete演算子を使って、指定した時点でメモリを解放します。

動的に確保したメモリを解放するには、次のように行います。

構文 delete演算子によるメモリの解放

```
delete ポインタ名;
```

つまり、メモリを利用し終えたら、コード中に、

```
delete pA;
```

メモリを解放します

と記述しなければなりません。なお、すでに解放したメモリに対してdelete演算子を使うことはできません。

これを行わないと、プログラムを実行するたびにメモリが確保され、利用できるメモリが少なくなっていってしまいます。

動的にメモリの確保・解放を行うと、指定した時点でメモリを利用することが可能になります。変数を利用した場合とくらべてみてください。

図10-8 動的に確保したメモリの寿命

それでは、メモリを動的に確保する場合の、一連の手順をコードでみてみることにしましょう。

Sample4.cpp ▶ 動的にメモリを確保する

```
#include <iostream>
using namespace std;

int main()
{
    int* pA;

    pA = new int;   ● メモリを確保します

    cout << "動的にメモリを確保しました。¥n";

    *pA = 10;

    cout << "動的に確保したメモリを使って" << *pA <<
       "を出力しています。¥n";

    delete pA;   ● メモリを解放します

    cout << "確保したメモリを解放しました。¥n";

    return 0;
}
```

Sample4の実行画面

```
動的にメモリを確保しました。
動的に確保したメモリを使って10を出力しています。
確保したメモリを解放しました。
```

ここでは、動的に確保したメモリを利用して値を記憶し、出力してみました。この小さなコードで利用する方法では、なかなか動的なメモリの確保の便利さは理解しにくいかもしれません。

しかし、たくさんの関数が登場する、大規模なコードの場合を考えてみてください。関数の開始・終了にかかわらず、記憶しておきたい値がある場合には、ローカル変数は使えません。ローカル変数の寿命は、その関数内だけに限られているからです。

また、グローバル変数では、プログラムの実行中はずっと値を保持することが

できます。しかし、グローバル変数は限りあるメモリを無駄づかいすることにもなります。

この2つの方法にくらべて、動的にメモリを確保する方法では、限りあるメモリを必要なときに十分に利用することができるようになります。

重要

動的なメモリはコード中の必要な部分でnew演算子により確保し、使い終わったらdelete演算子で解放する。

配列を動的に確保する

動的にメモリを確保することは、配列を扱う際に特に重要です。プログラムの実行時に配列の大きさを指定して、扱うことができるようになるからです。

第9章で説明したように、配列の要素数をいくつ利用するのかわからない場合は、配列の要素を大きめに確保しておかなければなりません。しかし、配列を動的に確保すれば、大きな配列を用意する必要はありません。配列のサイズをプログラムを実行する際に決めることができるのです。

まず、配列の動的な確保を行う構文をおぼえましょう。

構文 **動的な配列の確保**

```
ポインタ名 = new 型名[要素数];
```

この場合にも、配列を利用し終わったらメモリを解放する必要があります。このためには次のように[]をつけて解放します。

構文 **動的な配列の解放**

```
delete[] ポインタ名;
```

それでは、実際に配列を動的に確保してみることにしましょう。

Sample5.cpp ▶ 配列を動的に確保する

```
#include <iostream>
using namespace std;

int main()
{
    int num;
    int* pT;

    cout << "何人のテストの点数を入力しますか？¥n";

    cin >> num;

    pT = new int[num];

    cout << "人数分の点数を入力してください。¥n";

    for(int i=0; i<num; i++){
       cin >> pT[i];
    }

    for(int j=0; j<num; j++){
       cout << j+1 << "番目の人の点数は" << pT[j] << "です。¥n";
    }

    delete[] pT;

    return 0;
}
```

キーボードから数値を入力させ・・・

その分だけの配列要素を確保します

ポインタを使って点数を記憶します

Sample5 の実行画面

```
何人のテストの点数を入力しますか？
5 ↵
人数分のテストの点数を入力してください。
80 ↵
60 ↵
55 ↵
22 ↵
75 ↵
1番目の人の点数は80です。
2番目の人の点数は60です。
3番目の人の点数は57です。
```

Lesson 10

```
4番目の人の点数は22です。
5番目の人の点数は75です。
```

このコードでは、テストの点数を格納するための配列の要素数が決まっていません。要素数は、ユーザーがキーボードから要素数（人数）を入力してから決定しているのです。このように、入力された人数分の配列を動的に確保すれば、大きめの配列を用意する必要がなく、余分なメモリを使わずにすむのです。

図10-9 動的な配列の確保
ユーザーが入力した数値に応じて、配列を動的に確保することができます。

重要

プログラムの実行時に配列の大きさを決める場合には、配列を動的に確保する。

確保されるメモリ領域

この章では、さまざまなメモリの確保方法をみてきました。
ローカル変数や関数の引数が確保されるメモリ領域は**スタック**と呼ばれています。また、グローバル変数などの静的変数が確保される領域は**静的記憶領域**と呼ばれ、さらに、動的に確保されるメモリ領域は**動的記憶領域**（**ヒープ領域**）と呼ばれています。

10.4 ファイルの分割

ファイルを分割する

　この節では、大規模なプログラムを作成していく作業を学びましょう。大きなプログラムを作成するときには、数多くの関数を使うことになります。一度作成した便利な関数は、いろいろなプログラムから再利用することができれば便利です。そうすれば、これまでに作成した関数を使って、大きなプログラムをすぐに開発しやすくなるからです。

　このとき、何度も別のプログラムから利用するような関数は、main()関数と同じファイル内に記述するのではなく、別のファイルに分割して記述するのが普通です。ファイルを分割することで、いろいろなプログラムから利用しやすくするのです。

　次のコードはファイルを分割したプログラムです。

myfunc.h

```
//max関数の宣言
int max(int x, int y);
```

関数プロトタイプ宣言を記述します

ヘッダファイルです

myfunc.cpp

```
//max関数の定義
int max(int x, int y)
{
   if (x > y)
      return x;
```

関数の定義を別ファイルに記述します

```
    else
        return y;
}
```

Sample5.cpp ▶ ファイルを分割する

```
#include <iostream>
#include "myfunc.h"
using namespace std;

int main()
{
    int num1, num2, ans;

    cout << "1番目の整数を入力してください。¥n";
    cin >> num1;

    cout << "2番目の整数を入力してください。¥n";
    cin >> num2;

    ans = max(num1, num2);

    cout << "最大値は" << ans << "です。¥n";

    return 0;
}
```

ヘッダファイルを読み込みます

別ファイルの関数を呼び出します

Sample5の実行画面

```
1番目の整数を入力してください。
10 ↵
2番目の整数を入力してください。
5 ↵
最大値は10です。
```

このコードでは、ファイルを次の3つに分割しています。

myfunc.h ・・・ 関数プロトタイプ宣言

myfunc.cpp ・・・ 作成したmax()関数の定義

Sample5.cpp ・・・ main()関数の定義（プログラム本体）

この場合、まずSample5.cpp、myfunc.cppを別々にコンパイルしてオブジェクトファイルを作成します。さらにこのオブジェクトファイルどうしをリンクすることで、1つのプログラムを作成します。

myfunc.hというファイルには、関数の仕様をあらわす関数プロトタイプ宣言だけを記述しておきます。このように関数プロトタイプ宣言をまとめたファイルは**ヘッダファイル**（header file）と呼ばれています。

ソースファイル
myfunc.cpp

```
int max(int x, int y)
{
    ...
}
```

ソースファイル
Sample5.cpp

```
#include<iostream>
#include "myfunc.h"

int main()
{
    ans = max(num1, num2);
}
```

ヘッダファイル
iostream
myfunc.h

```
int max(int x, int y);
```

チェック
コンパイル
オブジェクトファイル
オブジェクトファイル
リンク
プログラム

図10-10 ファイルの分割
ファイルを分割してコンパイルすれば、大きなプログラムを効率よく作成することができます。

関数を利用するSample5.cppの先頭では、ヘッダファイルであるmyfunc.hを読み込むように指定しておきます。こうしておくと、ファイルを分割しても関数プロトタイプ宣言によって、コンパイルをするときに、関数の呼び出しが正しいかどうかをチェックすることができます。

複数のファイルのコンパイル・リンクの手順はお使いのC++開発環境によって異なりますので、対応する説明書を参考にしてみてください。

標準ライブラリのしくみを知る

ファイルを分割して大きなプログラムを作る方法を知ることができたでしょうか。ところで、C++の開発環境には、どのプログラムでも使える標準的な処理について、すでに定義された関数が添付されています。これらを、**標準ライブラリ**（standard library）といいます。これまで使ってきた入出力機能や文字列操作関数も標準ライブラリの関数なのです。

この関数を利用するには、標準のヘッダファイルをインクルードすることになっています。

標準のヘッダファイルをインクルードをするには、<> でかこみます。一方、自分で作成したヘッダファイルをインクルードするには" " でかこみます。

```
#include <iostream>
#include "myfunc.h"
```

通常、<> でかこんだファイルは標準ライブラリとして開発環境が指定しているディレクトリ（フォルダ）から読み込まれます。一方、" " でかこんだヘッダファイルは、自分の作成したソースファイルのあるディレクトリから読み込まれるようになっています。

Lesson 10

重要

標準ライブラリ関数を使ってコードを記述することもできる。

ヘッダファイル

iostream

ソースファイル

Sample5.cpp

```
#include <iostream>
int main(void)
{
    printf・・・
}
```

チェック

標準ライブラリ関数

コンパイル

オブジェクトファイル

オブジェクトファイル

リンク

プログラム

図10-11 標準ライブラリ関数
C++の環境に添付される標準ライブラリ関数を使って、コードを作成することができます。

スコープ

前の節で、変数のスコープ（通用範囲）について学んだことを思い出してください。

ファイルを分割した場合には、変数や関数が「どのファイルで通用するのか」ということも考慮しなければなりません。

グローバル変数と関数の名前はすべてのファイルで通用します。ただし、staticをつけると、ファイル内だけに限定することができます。

名前がすべてのファイルで適用することを「外部リンケージをもつ」といいます。また、名前がファイル内に限定されることを「内部リンケージをもつ」といいます。つまり、グローバル変数と関数は原則として外部リンケージをもちますが、staticをつけると内部リンケージをもつことになります。つまり、ファイルを分割した場合のスコープは次のようになります。

	記憶クラス指定子	スコープ
ローカル変数	（無指定）static	ブロック内
グローバル変数	（無指定）	すべてのファイル（外部リンケージ）
関数	static	ファイル内（内部リンケージ）

なお、別のファイルの（無指定の）グローバル変数を利用するときには、extern という指定をつけたヘッダファイルを読み込んでコンパイルします。

myfunc.cpp

```
int a;
void func()
{
}
```

Sample.cpp

```
#include myfunc.h
int main()
{
    a++;
}
```

myfunc.h

```
extern int a;
```

10.5 レッスンのまとめ

この章では、次のようなことを学びました。

- 変数などは、宣言位置によってグローバルとローカルに分類できます。
- グローバル変数は、すべての関数内から利用でき、静的寿命をもちます。
- ローカル変数は、定義を行った場所から関数終了まで利用できます。
- ローカル変数にstaticを指定すると静的寿命をもちます。
- new演算子でメモリを確保し、delete演算子で解放することができます。
- 大規模なプログラム作成する際には、ファイルを分割します。

この章では、大規模なコードを作成するために注意しておかなければならないことを学びました。複雑なコードを記述するためには、この章で学んだいろいろな知識を使わなければならないことがあります。この章の注意を念頭において、複雑なプログミングを行っていくことにしましょう。

練習

1. 次の項目について○か×で答えてください。

① グローバル変数とローカル変数には、同じ識別子を使うことができる。

② 別の関数内で宣言された２つのローカル変数には、同じ識別子を使うことができる。

③ 無指定のローカル変数は静的寿命をもつ。

④ ローカル変数はそれを宣言した関数外の関数から利用することができる。

⑤ 無指定のグローバル変数はどの関数からも利用することができる。

2. 次のコードの誤りを訂正してください。

```
#include <iostream>
using namespace std;

int main()
{
   int* pA;
   pA = new int;
   *pA = 10;

   return 0;
}
```

Lesson 11

いろいろな型

第3章では、C++に組み込まれている基本的な「型」について学びました。C++ではこのほかにも、さまざまな種類の型が存在します。この章では、私たちコードを作成する人間が取り決めることのできる特別な型について学びましょう。さまざまな型を使いこなすと、バリエーションにとむC++のプログラムを作成できるようになります。

Check Point!

- typedef
- 列挙
- 構造体
- メンバ
- ドット演算子（.）
- アロー演算子（->）
- 共用体

11.1 typedef

typedefのしくみを知る

第3章では、さまざまな型について学びました。この章ではまず、**typedef**というキーワードをとりあげることにしましょう。typedefは、いままで学んだint型やdouble型などの型に対して、新しい名前をつけるためのキーワードです。typedefは、次のように使います。

構文 **typedef**

```
typedef 型名 識別子;
```

型に独自の識別子をつけて記述することができます

次のコードをみてください。

```
typedef unsigned long int Count;
```

長い型名に短い名前をつけています

このコードは、unsigned long int型にCountという短い名前をつけたものです。このようにtypedefを使うと、次のように「Count」型の変数numを使うことができるようになります。

```
Count num = 1;
```

Count型の変数を使うことができます

この文は、実質的には、次のコードと同じ意味となります。

```
unsigned long int num = 1;
```

つまり、typedefはすでにある型に対し、別の名前をつけることができるのです。
typedefを使うと、長い型名に別名をつけることができるので、コードを読みやすくすることができます。

図11-1 typedef
独自の型名をつけるには、typedefを使います。

重要
typedefで独自の型名をつけることができる。

11.2 列挙

列挙型のしくみを知る

これまでに学んだint型やdouble型などの型は、C++にすでに組み込まれている型（基本型）でした。これに対し、私たちプログラムを作成する人間によって、

新しい型を作成する

ことができます。このような型を**ユーザー定義型**（user defined type）といいます。

まず、この節では**列挙型**（enumrated data type）について学びましょう。

新しい型を作成するには、まず、その型がどんな値をあらわすのかを取り決める必要があります。列挙型の場合は、**列挙型の宣言**によって、あらわす値を取り決めます。列挙型を宣言するには、**enum**というキーワードを使います。

構文 列挙型の宣言

```
enum 列挙型名 {識別子1, 識別子2, 識別子3, ･･･};
```

enumをつけて宣言します

列挙型は識別名を値として格納できる型のことをいいます。たとえば、次のようなものが列挙型です。

```
enum Week{SUN, MON, TUE, WED, THU, FRI, SAT};
```

Week型はこれらの値を格納します

この列挙型Weekは、SUN、MON ･･･ という識別子の値を格納できる型になります。

列挙型の変数を宣言する

それでは列挙型を使ってみましょう。列挙型を宣言すると、これを新しい型としてコード中で使うことができるようになります。たとえば、「Week型」の変数というものをコード中で宣言することができるようになるわけです。Week型の変数も、通常の変数と同じように宣言します。

構文　列挙型変数の宣言

```
列挙型名 列挙変数名;
```

列挙型の変数を宣言します

次のように宣言した変数wは、Week型の値を格納する変数となります。

```
Week w;
```

Week型の変数wです

では実際に、列挙型を宣言して利用してみることにしましょう。次のコードをみてください。

Sample1.cpp ▶ 列挙型の変数を使う

```
#include <iostream>
using namespace std;

//列挙型Weekの宣言
enum Week{SUN, MON, TUE, WED, THU, FRI, SAT};

int main()
{
   Week w;
   w = SUN;

   switch(w){
      case SUN: cout << "日曜です。¥n"; break;
      case MON: cout << "月曜です。¥n"; break;
      case TUE: cout << "火曜です。¥n"; break;
      case WED: cout << "水曜です。¥n"; break;
```

列挙型を宣言します

Week型の変数wを宣言します

値SUNを代入します

値により異なる出力を行います

```
        case THU: cout << "木曜です。¥n"; break;
        case FRI: cout << "金曜です。¥n"; break;
        case SAT: cout << "土曜です。¥n"; break;
        default: cout << "何曜日だかわかりません。¥n"; break;
    }

    return 0;
}
```

Sample1 の実行画面

```
日曜です。
```

このコードでは、列挙型であるWeek型の変数wを宣言しました。Week型の変数wにはSUN、MON・・・といった値を格納することができます。このコードでは、wの値によって異なる出力を行う処理をしています。

列挙型の値は識別子なので、読みやすいコードを記述できます。ここでは、日曜から土曜までの値を使っているコードだということがわかりやすくなっていますね。

図11-2 **列挙型**
列挙型の変数には識別子を格納することができます。

重要 列挙型を使って読みやすいコードを記述できる。

11.3 構造体

構造体型のしくみを知る

前の節では列挙型について学びました。さらにユーザー定義型をとりあげていくことにします。この節では**構造体型**（structure datatype）を学びます。

構造体型も列挙型と同じように、私たちが作成できるユーザー定義型のひとつです。構造体型は異なる型の値をまとめるための型となっています。たとえば、車のナンバー（int型）やガソリン量（double型）といった異なる型の値をまとめ、車をあらわす型とすることができるのです。

それでは実際に、構造体型の取り決めかたを説明しましょう。構造体型について取り決めることを、**構造体型の宣言**といいます。構造体型を宣言するには、**struct**というキーワードを使います。

構文　構造体型の宣言

```
struct 構造体型名 {    ← structをつけて宣言します
    型名 識別子;
    型名 識別子;
    ...
};
```

構造体型はブロックの中に変数などをまとめて記述したものです。たとえば、車のナンバーとガソリン量を管理するための構造体型Carの宣言は、次のようになります。

```
struct Car{          ← 「車」をあらわす構造体型です
   int num;          ← ナンバーを格納します
   double gas;       ← ガソリン量を格納します
};
```

Lesson 11

ここでは、int型の変数numとdouble型の変数gasを宣言しました。numは車のナンバーを、gasはガソリン量をあらわすためのものです。

重要 構造体型を宣言して、異なる型をまとめることができる。

図11-3 構造体型の宣言
異なる型の値をまとめて構造体型とすることができます。

構造体変数を宣言する

構造体型も新しい型としてコード中で使うことができるようになります。たとえば、「Car型の変数」というものをコード中で宣言することができるようになるわけです。Car型の変数も、通常の変数と同じように宣言します。

構文 構造体変数の宣言

```
構造体型名 構造体変数名;
```
構造体型の変数を宣言します

次のように宣言した変数car1は、構造体であるCar型の値を格納する変数となります。

```
Car car1;
```
Car型の値を格納する変数car1です

メモリ
car1
Car 型

図 11-4 構造体変数の宣言
構造体型の変数を宣言することができます。

メンバにアクセスする

構造体型の変数（構造体）を宣言すると、その中のnumやgasに、実際のナンバーやガソリン量の値を格納することができるようになります。このnumやgasといった変数のことを、**メンバ**（member）と呼んでいます。構造体のメンバを利用するためには、**ドット演算子**（.）というものを使います。メンバを利用することを、**メンバにアクセスする**ともいいます。

構文 構造体のメンバへのアクセス

```
構造体変数名.メンバ
```

たとえば、構造体型の変数car1の場合には、次のように代入をしてメンバに値を格納できるのです。

```
car1.num = 1234;
car1.gas = 25.5;
```

ナンバーをあらわすメンバnumに1234を代入します
ガソリン量をあらわすメンバgasに25.5を代入します

では、実際にコードを記述してみることにしましょう。

Sample2.cpp ▶ 構造体のメンバにアクセスする

```
#include <iostream>
using namespace std;

//構造体型Carの宣言
struct Car{
   int num;
   double gas;
};

int main()
{
   Car car1;

   cout << "ナンバーを入力してください。¥n";
   cin >> car1.num;

   cout << "ガソリン量を入力してください。¥n";
   cin >> car1.gas;

   cout << "車のナンバーは" << car1.num << ":ガソリン量は" <<
car1.gas << "です。¥n";

   return 0;
}
```

構造体型を宣言します

構造体型の変数を宣言します

メンバに値を代入します

メンバの値を出力します

Sample2 の実行画面

```
ナンバーを入力してください。
1234 ↵
ガソリン量を入力してください。
25.5 ↵
車のナンバーは1234:ガソリン量は25.5です。
```

このコードでは構造体型Carを宣言しています。次に、Car型の変数であるcar1を宣言しています。さらに、変数car1にドット演算子を使って各メンバにアクセスしています。各メンバに値を格納してから画面に出力する処理を行っています。

このように、構造体変数にドット演算子を使うと、構造体のメンバにアクセスし、値を格納することができるようになります。

重要

ドット演算子を使って、構造体の各メンバにアクセスできる。

図11-5 構造体メンバへのアクセス
Car型の変数car1の各メンバにアクセスして、値を格納することができます。

構造体を初期化する

次にもうひとつ、構造体について便利な書きかたをおぼえておくことにしましょう。さきほどは構造体を宣言したあとで、ドット演算子（.）を使ってメンバに値を代入する方法を紹介しました。

```
Car car1;

car1.num = 1234;
car1.gas = 25.5;
```


この2つの作業を同時にまとめて行うこともできます。これを**構造体の初期化**といいます。構造体の初期化は次のようにして行います。

```
Car car1 = {1234, 25.5};
```


変数を宣言するときに{}でかこんで値を記述すると、カンマで区切った順番にメンバに値が格納されます。構造体にあらかじめ初期値を与えるときには便利な記述ですので、おぼえておきましょう。

構文　構造体の初期化

```
構造体型名 構造体変数名 = {値, 値, ・・・};
```

構造体に代入する

さて、これまでは各メンバに対して代入演算子で値を格納するコードを紹介してきました。では、構造体変数そのものに対して、代入演算子を使うとどうなるのでしょうか？ 次のコードをみてください。

Sample3.cpp ▶ 構造体に代入する

```
#include <iostream>
using namespace std;

//構造体型Carの宣言
struct Car{
   int num;
   double gas;
};

int main()
{
   Car car1 = {1234, 25.5};
   Car car2 = {4567, 52.2};

   cout << "car1の車のナンバーは" << car1.num << "ガソリン量は" <<
car1.gas << "です。¥n";
   cout << "car2の車のナンバーは" << car2.num << "ガソリン量は" <<
```

```
car2.gas << "です。¥n";

   car2 = car1;

   cout << "car1をcar2に代入しました。¥n";

   cout << "car2の車のナンバーは" << car2.num << "ガソリン量は" <<
car2.gas << "です。¥n";

   return 0;
}
```

構造体どうしで代入します

Sample3の実行画面

```
car1の車のナンバーは1234ガソリン量は25.5です。
car2の車のナンバーは4567ガソリン量は52.2です。
car1をcar2に代入しました。
car2の車のナンバーは1234ガソリン量は25.5になりました。
```

代入した構造体のメンバの値となっています

ここでは、car1とcar2という2つの構造体を宣言しました。さらに、次のように代入しています。

```
car1 = car2;
```

構造体どうしで代入します

この代入は、

car2のメンバにcar1のメンバを1つずつコピーして値を格納する

ということを意味しています。つまり、このような代入を記述すると、car1のメンバであるnumとgasの値がcar2のメンバにコピーされるのです。この結果、car2のナンバーとガソリン量はcar1と同じ値となります。

構造体どうしでこのような代入ができますので、おぼえておいてください。

Lesson 11

1234
25.5
car1

1234
25.5
car2

図 11-6 **構造体の代入**
構造体に代入すると、メンバがそれぞれコピーされて値が格納されます。

重要

構造体に代入をすると、各メンバに値が格納される。

11.4 構造体の応用

引数として構造体を使う

構造体はさまざまなコードの中で使っていくことができます。この節では、構造体を応用するコードを作成していくことにしましょう。まず最初に、構造体を関数の中で利用する方法をみてみましょう。構造体は、関数の引数として用いることができます。次のコードをみてください。

Sample4.cpp ▶ 構造体を関数の引数として使う

```
#include <iostream>
using namespace std;

//構造体型Carの宣言
struct Car{
   int num;
   double gas;
};

//show関数の宣言
void show(Car c);    ● 構造体を引数にもつ関数です

int main()
{
   Car car1 = {0, 0.0};

   cout << "ナンバーを入力してください。¥n";
   cin >> car1.num;

   cout << "ガソリン量を入力してください。¥n";
   cin >> car1.gas;

   show(car1);    ● 構造体car1（の値）を渡します
```

```
    return 0;
}

//show関数の定義
void show(Car c)
{
    cout << "車のナンバーは" << c.num << "ガソリン量は" << c.gas << "で
す。¥n";
}
```

渡された構造体（の値）を・・・

出力します

Sample4の実行画面

```
ナンバーを入力してください。
1234 ↵
ガソリン量を入力してください。
25.5 ↵
車のナンバーは1234ガソリン量は25.5です。
```

第8章でも学んだように、引数は原則として値渡しで関数内に渡されるのでした。構造体を引数とした場合も「値」が渡されます。これは、

実引数の構造体のメンバの値がそれぞれコピーされて、関数の本体に渡される

ということを意味しています。つまり、ここでは構造体のメンバであるnumとgasの値がコピーされて関数に渡されるようになっているのです。

図11-8 引数と構造体
関数の引数として構造体を使うと、各メンバがコピーされて渡されます。

構造体へのポインタを引数に使う

構造体を引数として使うと、各メンバの値がコピーされて関数内に渡されます。ただし、メンバがたくさんある構造体を引数として使う場合には注意が必要です。関数を呼び出すたびに、たくさんのメンバがコピーされるため、関数の呼び出しに時間がかかってしまうことがあるのです。

このため、大きな構造体を関数の引数として扱う場合には、

構造体へのポインタを引数として使う

ことがあります。つまり、構造体型の変数のアドレスを使うようにするのです。

構造体へのポインタを関数の引数としておくと、アドレスを渡すだけで関数が呼び出されます。こうすると、大きな構造体の場合には、処理速度が向上する場合があります。なお、この場合は引数を実質的に参照渡しとすることになるので、渡した構造体のメンバの値を関数内で変更することができます。

```
int main()
{
    show(&car1);

    return 0;
}

void show(Car* pC)
{
}
```


図 11-9 引数と構造体へのポインタ
関数の引数として構造体へのポインタを使うと、アドレスが渡されます。

ただし、このような関数内では、構造体へのポインタを使って各メンバにアクセスする処理を記述しなければなりません。ポインタからメンバにアクセスする場合には、**アロー演算子**（->）を用いると便利です。

構文 **構造体型へのポインタからメンバにアクセス**

構造体へのポインタ->構造体メンバ

次のコードを入力してみましょう。今度は引数として構造体へのポインタを渡す関数を使ったものです。

Sample5.cpp ▶ 構造体へのポインタを関数の引数に使う

```
#include <iostream>
using namespace std;

//構造体型Carの宣言
struct Car{
   int num;
   double gas;
};

//show関数の宣言
void show(Car* pC);

int main()
{
   Car car1 = {0, 0.0};

   cout << "ナンバーを入力してください。¥n";
   cin >> car1.num;

   cout << "ガソリン量を入力してください。¥n";
   cin >> car1.gas;

   show(&car1);

   return 0;
}

//show関数の定義
void show(Car* pC)
{
   cout << "車のナンバーは" << pC->num << "ガソリン量は" << pC->gas
<< "です。¥n";
}
```

構造体へのポインタを引数にもつ関数です

構造体car1のアドレスを渡します

ポインタからメンバにアクセスします

show()関数内の処理をみてください。

```
   cout << "車のナンバーは" << pC->num << "ガソリン量は" << pC->gas
<< "です。¥n";
```

この関数内ではポインタが渡されるので、メンバにアクセスする場合にはドット演算子（.）ではなく、アロー演算子（->）を使っています。実行結果はSample4と同じです。

このコードのような小さな構造体では、引数がポインタでなくても、呼び出しの速度はほとんどかわりません。しかし、メンバをたくさんもつ大きな構造体を使う場合には、この差が無視できない場合もあるので、おぼえておくとよいでしょう。

重要

構造体へのポインタからメンバにアクセスするには、アロー演算子を使うと便利である。

構造体への参照を引数に使う

ところで、第8章では、ポインタのかわりに参照を引数として使う方法を学びました。構造体変数への参照を引数として使っても、Sample4と同じ効果が得られます。そこで、ここでも、参照を使った場合のコードを示しておくことにしましょう。

Sample6.cpp ▶ 参照を関数の引数に使う

```
#include <iostream>
using namespace std;

//構造体型Carの宣言
struct Car{
   int num;
```

```
    double gas;
};

//show関数の宣言
void show(Car& c);

int main()
{
    Car car1 = {0, 0.0};

    cout << "ナンバーを入力してください。¥n";
    cin >> car1.num;

    cout << "ガソリン量を入力してください。¥n";
    cin >> car1.gas;

    show(car1);

    return 0;
}

//show関数の定義
void show(Car& c)
{
    cout << "車のナンバーは" << c.num << "ガソリン量は" << c.gas <<
"です。¥n";
}
```

構造体への参照を引数にもつ関数です

構造体car1を渡します

渡された構造体で参照が初期化されます

参照からメンバにアクセスします

このコードの実行結果はSample4と同じです。ただし、このコードでは、関数内でメンバにアクセスする際、アロー演算子（->）ではなく、ドット演算子（.）を使っていることに注意してください。構造体型へのポインタからアクセスするときにだけ、アロー演算子を使うのです。

構造体の高度な機能

C++の構造体は、このほかにもいくつかの高度な機能をもっています。しかし、C++では、さらに便利な「クラス」（第12章）という機能が用意されています。構造体の高度な機能は、クラスを使って記述することが通常です。そこで、構造体の高度な機能は省略することにしましょう。

11.5 共用体

共用体型のしくみを知る

最後に、**共用体型**(union data type)という型について学んでおきましょう。共用体型は構造体型と似たスタイルをもつユーザー定義型です。共用体型の宣言は次のようになります。

構文 共用体型の宣言

```
union 共用体型名{
    型名 識別子;
    型名 識別子;
    ...
};
```

union をつけて宣言します

共用体型は、構造体型のstructキーワードのかわりにunionというキーワードを使います。

共用体型も、共用体型の変数を用意することによって値を格納することができるようになります。

ただし、共用体型の各メンバは、

同時に値を記憶できるのではなく、全体で1つしか値を記憶することができない

という特徴があります。次のコードをみてください。

Sample7.cpp ▶ 共用体型を使う

```
#include <iostream>
using namespace std;
```

```
//共用体型Yearの宣言
union Year{
   int ad;
   int gengo;
};

int main()
{
   Year myyear;

   cout << "西暦を入力してください。¥n";
   cin >> myyear.ad;

   cout << "西暦は" << myyear.ad << "です。¥n";
   cout << "元号も" << myyear.gengo << "です。¥n";

   cout << "元号を入力してください。¥n";
   cin >> myyear.gengo;

   cout << "元号は" << myyear.gengo << "です。¥n";
   cout << "西暦も" << myyear.ad << "です。¥n";

   return 0;
}
```


Sample7の実行画面

```
西暦を入力してください。
2000 ↵
西暦は2000です。
元号も2000です。
元号を入力してください。
12 ↵
元号は12です。
西暦も12です。
```


このコードでは、共用体型union Yearを宣言して新しい型としています。さらに、共用体型の変数myyearを宣言して、そのメンバにアクセスしています。

コードの実行結果を注意深くみてください。共用体のメンバadとgengoは

同時に別の値を記憶することができない

ということがわかるでしょう。あるメンバの値を変更すると、ほかのメンバの値も同じものに変更されています。共用体のメンバはすべてメモリの同じ位置を共有し、値を格納します。そのため、adに値を代入するとgengoの値はadと同じになってしまいます。また、gengoに値を代入すると、逆にadの値がgengoと同じになってしまいます。

このように、共用体は限られたメモリを節約するために使われる型となっています。

重要

共用体のメンバは、一度に1つだけしか値を記憶できない。

図11-10 共用体
共用体の各メンバは、全体で1つしか値を記憶することができません。

いろいろな型のまとめ

いままでに出てきたユーザー定義型を含めたいろいろな型をまとめておきましょう。基本型については第3章の表を参考にしてください。クラスについては、次章で学びます。

- 基本型
  - 論理型
  - 文字型
  - 整数型
  - 浮動小数点型
- ユーザー定義型
  - 列挙型
  - 構造体型
  - 共用体型
  - クラス
- ポインタ型
- 配列型
- 参照型

11.6 レッスンのまとめ

この章では、次のようなことを学びました。

- typedefを使って、独自の型名をつけることができます。
- 基本型以外に、ユーザー定義型を作成することができます。
- 列挙型は、識別子の値を格納することができます。
- 構造体型は、異なる型の値をまとめる型です。
- 構造体のメンバにアクセスするには、ドット演算子（.）を用います。
- 構造体のポインタからメンバにアクセスするには、アロー演算子（->）を用いると便利です。
- 共用体の各メンバは、同じメモリを共有しています。

この章で学んだように、C++では、さまざまな型を宣言し、利用することができます。特に構造体は、異なる型の値をまとめて格納できる便利な機能です。さまざまな型を定義する方法をおぼえておいてください。

練習

1. 構造体Person型を宣言し、人間の年齢（int型　age）・身長（double型　height）・体重（double型　weight）を管理するコードを作成してください。実際に2人の人物の年齢、身長、体重を入力し、次のように出力するコードを記述してください。

```
年齢を入力してください。
28 ↵
身長を入力してください。
165.3 ↵
体重を入力してください。
52.2 ↵
年齢を入力してください。
32 ↵
身長を入力してください。
168.8 ↵
体重を入力してください。
62.5 ↵
年齢28身長165.3体重52.2です。
年齢32身長168.8体重62.5です。
```

2. 1の構造体へのポインタを引数として受けとり、年齢を1つ増やす関数void aging(Person* p)を作成してください。実際に1人の人物の情報を入力し、1年後の年齢を出力するコードを記述してください。

```
年齢を入力してください。
28 ↵
身長を入力してください。
165.3 ↵
体重を入力してください。
52.2 ↵
年齢28身長165.3体重52.2です。
1年経過しました。
年齢29身長165.3体重52.2です。
```

Lesson 12

クラスの基本

これまでの章では、変数・配列など、さまざまなC++の機能について学んできました。このような機能は、以前から多くのプログラミング言語に組み込まれてきたものです。しかし、プログラムが複雑なものになるにつれて、効率よくプログラムを作成するしくみが必要になってきました。こうしてとり入れられた新しい機能が「クラス」です。この章では、クラスの基本を学ぶことにしましょう。

Check Point!

- クラス
- オブジェクト
- スコープ解決演算子
- データメンバ
- メンバ関数
- privateメンバ
- publicメンバ

12.1 クラスの宣言

クラスのしくみを知る

これまでの章では、変数・配列・関数など、さまざまなC++言語の機能を学んできました。このような機能は、C++言語ばかりでなく、多くのプログラミング言語に組み込まれています。C++は、C言語からこれらの機能を受け継いでいます。

しかし、プログラムの作成が複雑になるにつれて、プログラムを限られた期間で効率よく作成するための機能が重要になってきました。この新しい機能のひとつが、**クラス**（class）という概念です。クラスの機能を利用すると、複雑なプログラムを効率よく作成することができます。

この章からは、クラスの強力な機能を学んでいくことにします。最初に、「クラス」とはどういうものなのか、かんたんにながめておきましょう。

クラスを扱うときには、**現実の世界に存在する、「モノ」の概念に着目**していきます。たとえば、「車」といったモノに着目し、プログラムを作成することを考えてみることにします。車は、1234や4567といったナンバーをもち、20.5リッターや30.5リッターといったガソリンを積んでいますね。このとき、「車」に関するデータとして、次のようなものを考えることができるでしょう。

- ナンバー
- ガソリンの残量

「ナンバーは○○である」「ガソリンの残量は○○である」といった「車」一般に関することがらを「車クラス」にとり入れていくことにするのです。これらのデータは、いわば車の「性質」のようなものだと考えることもできます。

また、このほかにも車には、次のような「機能」があると考えられます。

- 車のナンバーを決める
- 車にガソリンを入れる

■ 車のナンバーやガソリン量を表示する

これらの「機能」は、車のナンバーやガソリン量などを変更したりするものといえます。クラスとは、このような

モノの性質や、それにかかわる機能をまとめながら、プログラムを作成していく

ために使う概念なのです。

クラスをまとめるには、おおよそ次のようなコードを記述します。これがクラスの基本です。

```
//車クラス
class 車 {
    ナンバー;                          ┐
    ガソリン量;                        ┘── 車の性質をまとめます
    ナンバーを決める・・・               ┐
    ガソリンを入れる・・・               ├── 車の機能をまとめます
    ナンバーとガソリン量を表示する・・・ ┘
};
```

ブロックの中に、車の「性質」や「機能」をまとめました。これに「車」という名前をつけています。

```
class 車 {
  ナンバー
  ガソリン量
  ナンバーを決める
  ガソリンを入れる
  ナンバーとガソリン量を表示する
  ・・・
};
```

図12-1 クラス
モノ一般に関する性質や機能をまとめたものをクラスといいます。

クラスを宣言する

それでは、クラスの書きかたを学ぶことにしましょう。実は、クラスの書きかたは、第11章で学んだ構造体型とほとんど同じです。C++のクラスとは、私たちが作成できる**ユーザー定義型**のひとつとなっています。

クラスにモノの性質や機能をまとめることを、**クラスを宣言する**といいます。クラスは次のように宣言します。

構文　クラスの宣言

```
class クラス名 {
    アクセス指定子:
    変数の宣言;
    ...
    関数の宣言;
    ...
};
```


構造体の場合はstructキーワードを使いましたが、クラスの場合は、classキーワードを使います。

クラス内には、変数や関数をまとめて宣言します。この変数や関数は、構造体と同じように**メンバ**(member)と呼ばれています。特にクラスの場合は、変数を**データメンバ**(data member)、関数を**メンバ関数**(member function)と呼んでいます。

ただし、メンバ関数については注意が必要です。前のページの構文では、メンバ関数の宣言のみを行っています。つまり、メンバ関数の実際の処理については定義していません。そこで、メンバ関数の本体については、クラスの宣言の外側で定義することにします。

構文　メンバ関数の定義

```
戻り値の型 クラス名::メンバ関数名(引数リスト)
{
    ...
}
```

::は**スコープ解決演算子**（scope resolution operator）といいます。::演算子を使って、メンバがどのクラスのメンバ関数であるのかを指定するのです。

重要

クラスはデータメンバとメンバ関数をもつ。

構造体とクラス

第11章では構造体について学びました。実は、C++では、構造体・クラスのどちらも、変数・関数をひとまとめにする機能をもっています。しかし、関数をまとめる場合には、構造体を使わずにクラスを使うことが多くなっています。

クラスを宣言するコード

それでは、実際にクラスを宣言してみましょう。次のコードは、車のナンバーとガソリン量を管理し、出力する機能をもつ「Car（車）」クラスを宣言したものです。

簡単なクラスの宣言

```
//Carクラスの宣言
class Car {
   public:
      int num;
      double gas;
      void show();
};

//Carクラスメンバ関数の定義
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
```

```
}
```

このCarクラスは、次のようなデータメンバとメンバ関数をもっています。

データメンバ	説明
num	ナンバーを格納する変数
gas	ガソリン量を格納する変数
メンバ関数	**説明**
show()	ナンバーとガソリン量を出力する関数

2つのメンバ関数の定義は、クラスの宣言の外側に記述されています。「Car::」という指定をつけることで、通常の関数ではなく、Carクラスのメンバ関数であることを示しています。

このように、クラスは、

データメンバ・・・モノの性質
メンバ関数・・・モノの機能

をまとめるのです。

```
class Car {
  public:
    int num;
    double gas;
    void show();
};
```

図12-2 クラスを宣言する
クラスはデータメンバとメンバ関数をまとめます。

クラスを利用する

さて、すでに説明したように、宣言したクラスは、新しい型のひとつとなりま

す。つまり、クラス型の変数を宣言することができるのです。クラス型の変数は、次のように宣言します。

構文 オブジェクトの作成

```
クラス名 変数名;
```
クラスの値を格納する変数を宣言します

クラスは新しい型ですから、通常の変数を宣言する方法と同じです。

それでは、Carクラスの変数を宣言するコードをみてみましょう。

```
Car car1;
```
Carクラスの値を格納する変数car1を宣言します

これは、「Carクラスの値」を記憶できる変数car1の宣言です。このクラス型の変数は、**オブジェクト**（object）または**インスタンス**（instance）と呼ばれます。本書では、クラス型の値を格納する変数のことをオブジェクトと呼ぶことにします。

重要 オブジェクトとは、クラス型の値を格納する変数のことである。

メンバにアクセスする

それでは、上で宣言したオブジェクトをコード中で利用していくことにしましょう。

オブジェクトを利用するには、オブジェクトの各メンバにアクセスして値を格納する必要があります。このためには、構造体と同じようにドット演算子（.）を使います。

```
car1.num = 1234;
car1.gas = 20.5;
```
ナンバーを1234とします
ガソリン量を20.5とします

また、メンバ関数の呼び出しを行う場合にもドット演算子を使います。

```
car1.show();
```

メンバ関数を呼び出してナンバーとガソリン量を出力します

それでは、クラスを宣言し、実際に利用するコードを入力してみましょう。

Sample1.cpp ▶ クラスを使う

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   public:
      int num;
      double gas;
      void show();
};

//Carクラスメンバ関数の定義
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
   cout << "ガソリン量は" << gas << "です。¥n";
}

int main()
{
   Car car1;

   car1.num = 1234;
   car1.gas = 20.5;

   car1.show();

   return 0;
}
```

- Carクラスの宣言です
- メンバ関数本体の定義です
- クラス型の変数（オブジェクト）を定義します
- ナンバーとガソリン量をデータメンバに代入します
- メンバ関数を呼び出してナンバーとガソリン量を出力します

Sample1 の実行画面

```
車のナンバーは1234です。
ガソリンの量は20.5です。
```

このコードでは、前半部分でCarクラスの宣言を行い、後半部分（main()関数）でCarクラスからオブジェクトを宣言しました。実行結果からわかるように、car1というオブジェクトを宣言することによって、ナンバーやガソリン量をあらわすデータメンバに値を代入し、出力することができました。

つまり、car1を宣言することによって、コード上であたかも次のようなことを行ったことになります。

1台の「車」（car1）を作成した（オブジェクトを宣言した）

↓

ナンバーとガソリン量を設定した（データメンバに値を代入した）

↓

ナンバーとガソリン量を表示させた（メンバ関数を呼び出した）

「車」の概念を中心にプログラムを組み立てているようすがわかるでしょうか？

オブジェクトを作成する

なお、オブジェクトのためにメモリを確保すること（ここでは、car1を宣言すること）を

オブジェクトを作成する

といいます。

逆に、オブジェクトを利用し終わって、メモリが解放されることを

オブジェクトを破棄する

といいます。Sample1で作成したオブジェクトcar1は、main()関数内で宣言されたローカル変数です。したがって、main()関数が開始するときにオブジェクトが作成され、main()関数が終了したとき破棄されます。1台の車が作成され、破棄されるイメージをもってみてください。

また、オブジェクトは動的に作成することもできます。new演算子を使ってオブジェクトのためにメモリを確保し、delete演算子を使ってメモリを解放するのです。

```
{
    Car* pCar;              ← Carクラスへのポインタを準備します

    pCar = new Car;         ← オブジェクトを動的に作成してポインタにアドレスを代入します

    pCar->num = 1234;       ┐
    pCar->gas = 20.5;       ┘ アロー演算子を使ってメンバにアクセスします

    delete pCar;            ← オブジェクトを破棄します
}
```

Carクラスへのポインタ pCarを用意し、動的に確保したメモリのアドレスを格納しています。ポインタからメンバにアクセスするには、構造体と同じように**アロー演算子**(->)が使えます。

図12-3 オブジェクトの作成
オブジェクトのためにメモリが確保されることを「オブジェクトを作成する」ともいいます。

2つ以上のオブジェクトを作成する

また、Sample1ではオブジェクトを1つしか作成しませんでしたが、**オブジェクトはいくつでも作成することができます**。たとえば、2台の車を作成するには、変数car1のほかにcar2を宣言すればよいのです。

```
Car car1;             ● 1つ目のオブジェクトを作成します
car1.num = 1234;      ┐ 1台目の車のナンバーとガソリンです
car1.gas = 20.5;      ┘

Car car2;             ● 2つ目のオブジェクトを作成します
car2.num = 2345;      ┐ 2台目の車のナンバーとガソリンです
car2.gas = 30.5;      ┘
```

すると、2台の「車」は、それぞれの独自のナンバーやガソリン量をもつことになります。なお、car1、car2は変数名ですから、識別子の中から別の適当な名前をつけてもかまいません。

オブジェクトを複数作成すれば、さらに複雑なプログラムを作成することができるでしょう。

図12-4　2つ以上のオブジェクトの作成
複数のオブジェクトを作成することができます。

クラスを利用する手順のまとめ

さて、この節でみてきたように、クラスを利用するプログラムを作成するには、通常、次の2つの手順をふむ必要があることがわかります。

1. クラスを宣言する

2. クラスからオブジェクトを作成する

1の「クラスを宣言する」という作業は、

「車に関する一般的な仕様」(クラス) を設計する

という作業だと考えられます。

一方、2の「オブジェクトを作成する」という作業は、その仕様 (クラス) をもとにして、

「個々の車」(オブジェクト) を作成し、データを記憶したり操作したりする

という作業だと考えられます。

ここでは、1と2を続けて同じファイルに記述しました。しかし、この2つのコードを2つのファイルに分割し、別々の人間が記述することもできます。

1の段階でCarクラスをうまく設計しておくと、たいへん便利です。そのCarクラスを利用すれば、「車」を扱うさまざまなプログラムを、別の人間が効率よく作成していけるようになるからです。

大規模なプログラムを作成する場合には、1と2の作業が別の人間によって行われます。このようにプログラムの工程が分かれている場合には、まちがいのおきにくいプログラムを作成していくことが重要となります。クラスには誤りをおきにくくするための機能が用意されています。次の節でみていくことにしましょう。

12.2 メンバへのアクセスの制限

メンバへのアクセスの制限

Sample1では、データメンバにナンバーやガソリン量の値を代入しています。これによって、あたかも

実際の車にナンバーやガソリンの量を設定する

かのような処理が行われているわけですね。車のナンバーを「1234」、ガソリン量を「20.5」と設定しているわけです。

ところが、実はこれだけでは問題がおきてしまう場合があります。たとえば、Sample1のmain()関数の中では、次のような記述ができてしまうことに注意してください。

```
int main()
{
   Car car1;

   car1.num = 1234;
   car1.gas = -10.0;    // 誤ったガソリン量を代入しています

   car1.show();
}
```

このコードは何を意味しているのでしょうか？　これまでと同じように考えると、このコード中では、

car1車のガソリン量を-10とする

ということが行われているわけです。

けれども、これはおかしな話です。本当の車では「ガソリンの量をマイナスにする」ことはありません。

クラスは「モノ」の概念を中心に設計しています。ですから、複雑なプログラムを作成していくうちに、このような不自然な操作を行うことで、プログラムに誤りがおこってしまう場合があるのです。

通常クラスを設計するときには、このような不具合がおこらないように、さまざまなしくみを使っていきます。これから、そのしくみをひとつずつみていくことにしましょう。

図12-5 メンバへのアクセス
クラスの外から勝手にメンバにアクセスできると、誤りのおきやすいプログラムになってしまう場合があります。

privateメンバをつくる

ではSample1で、ガソリンがマイナスになってしまう誤りの原因は何なのでしょうか？ この原因は、

メンバを無制限に利用し、勝手な値（ここでは-10）を代入してしまったこと

にあるといえます。C++ではこのようなまちがいがおこらないように、

クラスの外から勝手にアクセスできないようなメンバとしておく

ことができます。このメンバのことを、**private**メンバといいます。ではさっそく、車のナンバーとガソリン量をprivateメンバにしてみることにしましょう。

```
class Car{
   private;
      int num;
      double gas;
      ...
};
```

データメンバはprivateメンバにしました

ここでは、メンバにprivateという指定をつけました。こうすると、Carクラスの外（main()関数）からデータメンバにアクセスすることができなくなるのです。

```
int main()
{
      ...
   //このようなアクセスはできなくなります。
   //car1.num = 1234;
   //car1.gas = -10.0;
}
```

クラスの外からprivateメンバにはアクセスできません

これで、車のガソリン量にマイナスの値が代入されることはなくなりました。

```
class Car{
   private;
      int num;      ✗
      double gas;   ✗
      ...
};
```

```
int main()
{
   ...
   //car1.num = 1234;
   //car1.gas = -10.0;
   ...
}
```

図12-6 **privateメンバ**
privateメンバにすると、クラスの外から勝手にアクセスできなくなります。

重要

privateメンバにはクラスの外からアクセスできない。

publicメンバをつくる

いまみたように、データメンバをprivateメンバにすると、クラスの外からアクセスできなくなります。

しかしこのようにすることで、本当にmain()関数の中で車のナンバーやガソリン量を設定することができなくなってしまうのでしょうか？

実は、アクセスする方法があります。今度は次のようなコードを入力してみてください。これは、Sample1を改良したコードです。

Sample2.cpp ▶ メンバへのアクセスを制限する

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
   public:
      void show();
      void setNumGas(int n, double g);
};

//Carクラスメンバ関数の定義
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
   cout << "ガソリン量は" << gas << "です。¥n";
}
void Car::setNumGas(int n, double g)
{
   if(g > 0 && g < 1000){
      num = n;
      gas = g;
      cout << "ナンバーを" << num << "にガソリン量を" << gas <<
"にしました。¥n";
   }
   else{
      cout << g << "は正しいガソリン量ではありません。¥n";
      cout << "ガソリン量を変更できませんでした。¥n";
```

データメンバはprivateにしました

メンバ関数はpublicにしました

渡された値を調べて・・・

正しければ値を設定します

誤った値を設定できないようにしています

```
    }
}

int main()
{
    Car car1;

    //このようなアクセスはできなくなります。
    //car1.num = 1234;
    //car1.gas = 20.5;

    car1.setNumGas(1234, 20.5);
    car1.show();

    cout << "正しくないガソリン量を（-10.0）を指定してみます・・・。¥n";
    car1.setNumGas(1234, -10.0);
    car1.show();

    return 0;
}
```

privateメンバにはアクセスできません

必ずpublicメンバを呼び出して値を設定します

Sample2の実行画面

```
ナンバーを1234にガソリン量を20.5にしました。
車のナンバーは1234です。
ガソリン量は20.5です。
正しくないガソリン量（-10.0）を指定してみます・・・。
-10.0は正しいガソリン量ではありません。
ガソリン量を変更できませんでした。
車のナンバーは1234です。
ガソリン量は20.5です。
```

誤った値は設定されません

Lesson 12

このコードでは、ナンバーとガソリン量を設定するために、setNumGas()というメンバ関数を新しく追加しました。ここではガソリン量が正しいかどうかをチェックしてから、データメンバに値を代入する処理をしていることに注意してください。

Carクラスの外から、ナンバーやガソリン量を直接設定することはできません。しかし、そのかわりに、setNumGas()メンバ関数ならば呼び出すことができます。このメンバ関数を使えば、正しい値であるかどうか必ずチェックされてから、ガ

ソリン量が設定されます。つまり、誤ったガソリン量が設定されることがなくなるのです。

setNumGas()メンバ関数には、**public**という指定をつけています。このメンバを、**publicメンバ**といいます。publicをつけたメンバは、クラスの外から利用することができます。こうして、privateとpublicを使いわけることによって、正しいナンバーやガソリン量を設定することができるのです。

重要

publicメンバにはクラスの外からアクセスできる。

```
class Car {
   ...
   public;
      void setNumGas(int n, double g);
      void show();
   {
     ...
   }
}
```

```
int main()
{
   ...
   car1.setNumGas(1234, 20.5);
   car1.show();
   ...
}
```

図12-7 publicメンバ
publicメンバにすると、クラスの外からアクセスすることができます。

カプセル化のしくみを知る

Sample2では、Carクラス自身にガソリン量が正しいかどうかをチェックする機能をもたせることができるようになりました。こうすれば、誤った値が設定されないクラスを設計できます。

第8章でも説明したように、クラスを扱うプログラムでは、クラスの仕様部分（クラス宣言）とクラスを利用する部分（main()関数など）を、異なる人が記述する場合があります。クラスを設計する人が、メンバを適切にprivateメンバとpublicメンバに分類しておけば、あとからほかの人間がそのクラスを利用した場合に、誤りのおきにくいプログラムを作成できるようになるので、たいへん便利です。

このように、クラスの中にデータ(データメンバ)と機能(メンバ関数)をひとまとめにし、保護したいメンバにprivateをつけて勝手にアクセスできなくする機能を**カプセル化**(encapsulation)といいます。一般的にはSample2のように、

データメンバ ⟶ privateメンバ
メンバ関数 ⟶ publicメンバ

と指定することがよく行われています。カプセル化は、クラスがもつ重要な機能のひとつです。

重要

データと機能をひとまとめにし、メンバを保護する機能をカプセル化という。

public
private

図12-8 カプセル化
クラスにカプセル化の機能をもたせることによって、誤りのおきにくいプログラムを作成できます。

publicとprivateを省略すると?

privateとpublicは**アクセス指定子**(access specifier)と呼ばれています。アクセス指定子は省略することもできます。省略した場合は、すべてprivateメンバとなります。

なお、C++では、第11章で解説した構造体の各メンバにも、アクセス指定子を指定することができます。ただし、構造体の場合には、省略した場合すべてpublicとなります。第11章ではすべて省略しているのでpublicメンバというわけです。

アクセス指定子には、このほかprotectedというキーワードがあります。protectedについては第14章でくわしく学びます。

メンバ関数をインライン関数にする

ここで、ひとつ注意をしておきましょう。

privateメンバにアクセスするには、必ずpublicメンバ関数を呼び出してアクセスしなければなりません。このため、このようなクラスを利用するときには、**ひんぱんにメンバ関数が呼び出される**可能性があります。しかし、第7章でのべたように、関数をひんぱんに呼び出すと実行速度が遅くなる場合があります。そこで、簡単なメンバ関数については、インライン関数とすることがよく行われます。次のコードをみてください。

```
//Carクラスの宣言
class Car {
   private:
      int num;
      double gas;
   public:
      int getNum(){return num;}
      double getGas(){return gas;}
      void show();
      void setNumGas(int n, double g);
};

//Carクラスメンバ関数の定義
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
}
...
```

メンバ関数の本体をクラス内で定義すると、インライン関数になります

クラスの外で定義しているメンバ関数は通常の関数になります

このコードは、Carクラスのコードの一部です。第7章でみたように、関数をインライン関数にする場合はinlineを指定します。しかし、メンバ関数の場合は、もっと簡単にインライン関数を定義できます。

クラス宣言内に関数本体を定義すると、自動的にインラインとしたことになる

のです。

上のコードのgetNum()関数、getGas()関数はクラス宣言内で定義されています

から、自動的にインライン関数となります。一方、クラス宣言の外で定義したshow()関数は、通常どおりの関数となります。

重要

クラス宣言内に定義したメンバ関数は、インライン関数となる。

メンバ関数の機能

メンバ関数は関数の一種ですから、第７章で説明したことがそのままあてはまります。すなわち、メンバ関数をオーバーロードしたり、デフォルト引数を指定したりすることが可能です。第13章では、特殊なメンバ関数のオーバーロードについて説明します。

12.3 引数とオブジェクト

引数としてオブジェクトを使う

次に、関数とオブジェクトの関係をみていきましょう。関数の引数として、オブジェクトを使ってみることにします。次のコードを入力してみてください。

Sample3.cpp ▶ 引数としてオブジェクトを使う

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
   public:
      int getNum(){return num;}
      double getGas(){return gas;}
      void show();
      void setNumGas(int n, double g);
};

//Carクラスメンバ関数の定義
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
   cout << "ガソリン量は" << gas << "です。¥n";
}
void Car::setNumGas(int n, double g)
{
   if(g > 0 && g < 1000){
      num = n;
      gas = g;
      cout << "ナンバーを" << num << "にガソリン量を" << gas <<
 "にしました。¥n";
```

```
    }
    else{
        cout << g << "は正しいガソリン量ではありません。¥n";
        cout << "ガソリン量を変更できませんでした。¥n";
    }
}

//buy関数の宣言
void buy(Car c);

int main()
{
    Car car1;

    car1.setNumGas(1234, 20.5);

    buy(car1);     ← 関数にオブジェクトの値を渡して呼び出します

    return 0;
}

//buy関数の定義
void buy(Car c)
{
    int n = c.getNum();     ← 渡されたオブジェクトの値を利用します
    double g = c.getGas();
    cout << "ナンバー" << n << "ガソリン量" << g <<
  "の車を購入しました。¥n";
}
```

Sample3の実行画面

```
ナンバーを1234にガソリン量を20.5にしました。
ナンバー1234ガソリン量20.5の車を購入しました。
```

このように、オブジェクトを引数として使うと、各メンバの値がコピーされて関数内に渡されます。つまり、ここではオブジェクトのメンバであるnumとgasの値がコピーされて関数に渡されるようになっているのです。これは、構造体を引数として使った場合と同じです。

引数としてオブジェクトへのポインタを使う

なお、オブジェクトの場合も、たくさんのメンバをもつときには、関数呼び出しが遅くなることがあります。

このような場合には、引数として、オブジェクトへのポインタを利用することになります。引数にポインタを使うと、**オブジェクトのアドレスだけが渡される**ので、メンバごとのコピーが行われず、大きなクラスの場合には、処理速度が向上することがあるからです。

次のコードでは、Sample3のbuy()関数の引数を、Carクラスのオブジェクトへのポインタとして定義しました。クラスの宣言とクラスのメンバ関数の定義部分はSample4と同じですので省略しています。

Sample4.cpp ▶ 引数としてポインタを使う

```
...
//buy関数の宣言
void buy(Car* pC);

int main()
{
   Car car1;

   car1.setNumGas(1234, 20.5);

   buy(&car1);

   return 0;
}

//buy関数の定義
void buy(Car* pC)
{
   int n = pC->getNum();
   double g = pC->getGas();
   cout << "ナンバー" << n << "ガソリン量" << g <<
 "の車を購入しました。¥n";
}
```

ポインタを引数とした関数です（void buy(Car* pC);）

ポインタを引数とした関数です（void buy(Car* pC)）

アロー演算子を使ってメンバにアクセスします（int n = pC->getNum();）

このコードでは、car1のアドレスを渡して関数を呼び出しています。関数内ではアロー演算子を使ってメンバにアクセスします。実行結果はSample4と同じです。

また、同じ効果は、参照を引数とすることでも得ることができます。次のコードは、Sample3のbuy()関数をCarクラスのオブジェクトへの参照として定義したものです。関数内ではドット演算子を使ってメンバにアクセスします。

Sample5.cpp ▶ 引数として参照を使う

```
...
//buy関数の宣言
void buy(Car& c);

int main()
{
   Car car1;

   car1.setNumGas(1234, 20.5);

   buy(car1);

   return 0;
}

//buy関数の定義
void buy(Car& c)
{
   int n = c.getNum();
   double g = c.getGas();
   cout << "ナンバー" << n << "ガソリン量" << g <<
 "の車を購入しました。¥n";
}
```

参照を引数とした関数です

ドット演算子を使ってメンバにアクセスします

```
int main()
{
   Car car1;

   buy(&car1);

   return 0;
}
void buy(Car* pC)
{

}
```


図12-9 関数の引数としてポインタを使う
メンバを多くもつ場合には、引数にポインタを使うと、関数の呼び出しが速くなることがあります。

12.4 レッスンのまとめ

この章では、次のようなことを学びました。

- クラスは、データメンバとメンバ関数をメンバとしてもちます。
- クラスは、ユーザー定義型のひとつとして利用することができます。
- private メンバは、クラスの外からアクセスすることはできません。
- public メンバは、クラスの外からアクセスすることができます。
- メンバ関数をクラス宣言外部で定義するには、スコープ解決演算子（::）を使います。
- メンバ関数をクラス宣言内で定義すると、インライン関数になります。

この章では、クラスの設計と簡単な利用について学びました。クラスを適切に設計すると、エラーのおきにくいプログラムを効率よく作成することができます。次の章ではさらにクラスについて学んでいきましょう。

練習

1. 次の項目に○か×で答えてください。

① クラス宣言内において、アクセス指定子を省略したメンバは、public メンバとなる。

② クラス宣言内において、メンバ関数を定義すると、インライン関数となる。

③ データメンバを public として指定することはできない。

④ public メンバはクラスの外側からアクセスすることができる。

⑤ private メンバはクラスの外側からアクセスすることができない。

2. 次のように、整数値をあらわす Point クラスを設計してください。

データメンバ

x：X 座標(ただし 0 ～ 10 とする)
y：Y 座標(ただし 0 ～ 10 とする)

メンバ関数

void setX(int a)：X 座標を設定する(範囲外の値の場合は 0 とする)
void setY(int b)：Y 座標を設定する(範囲外の値の場合は 0 とする)
int getX()：X 座標値を得る
int getY()：Y 座標値を得る

3. 2 のクラスを利用し、次のように出力するコードを記述してください。

```
X座標を入力してください。
3 ↵
Y座標を入力してください。
5 ↵
入力した座標は(3,5)です。
```

Lesson 13

クラスの機能

第12章では、かんたんなクラスの宣言と、その利用方法について学びました。クラスはこのほかにも、さまざまな機能をもっています。この章ではさらにくわしく、クラスの機能についてみていきましょう。

Check Point!

- コンストラクタ
- コンストラクタのオーバーロード
- デフォルトコンストラクタ
- 静的メンバ

13.1 コンストラクタの基本

コンストラクタのしくみを知る

第12章では、クラスからオブジェクトを作成し、利用する方法をみてきました。オブジェクトを作成する際には、さまざまな**初期化処理**が必要となると考えられます。たとえば、これまで作成してきた車を管理するプログラムを考えてみてください。車を作成したときには、ナンバーやガソリン量の初期値として0を代入するといった初期化処理が必要になると考えられます。

クラスを使ったプログラミングでは、この初期化処理を、**コンストラクタ**（constructor）と呼ばれる特殊なメンバ関数の処理として記述します。

コンストラクタは、

クラスからオブジェクトが作成されるときに、自動的に呼び出される

という特殊なメンバ関数となっています。コンストラクタの定義をみてみましょう。

構文　コンストラクタの定義

```
クラス名::クラス名(引数のリスト)
{
    ...
}
```

コンストラクタはクラス名を関数名として使います

コンストラクタは戻り値がありません。voidとも記述しません

コンストラクタは、必ずクラス名を関数名として用いることになっています。また、コンストラクタは戻り値をもちません。

Car クラス

Car()

図 13-1 コンストラクタ
オブジェクトが作成されるときに自動的に呼び出される特殊なメンバ関数をコンストラクタといいます。

コンストラクタの定義は、次のようになります。

コンストラクタには戻り値がありません

クラス名を関数名とします

```
Car::Car()
{
    num = 0;
    gas = 0.0;
    cout << "車を作成しました。¥n";
}
```

Carクラスからオブジェクトが作成されると、定義しておいたコンストラクタが自動的に呼び出され、処理が行われるようになっています。このCarクラスのコンストラクタではナンバーとガソリン量を0に設定しています。

それでは、実際にコンストラクタのはたらきをみてみることにしましょう。

Sample1.cpp ▶ コンストラクタを定義する

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
   public:
      Car();
      void show();
```

```
};

//Carクラスメンバ関数の定義
Car::Car()
{
    num = 0;
    gas = 0.0;
    cout << "車を作成しました。¥n";
}
void Car::show()
{
    cout << "車のナンバーは" << num << "です。¥n";
    cout << "ガソリンの量は" << gas << "です。¥n";
}

int main()
{
    Car car1;

    car1.show();

    return 0;
}
```

コンストラクタの定義です

オブジェクトを作成するとコンストラクタが呼び出されます

Sample1 の実行画面

```
車を作成しました。
車のナンバーは0です。
ガソリン量は0です。
```

コンストラクタ内の処理が行われました

ここでは、Carクラスからcar1というオブジェクトを作成しました。このとき、定義しておいたコンストラクタ（Car()）が自動的に呼び出され、「車を作成しました。」と出力が行われているのです。ナンバーとガソリン量も0に設定されていますね。

```
class Car {
   ...
  public:
    Car();
     ...
}
```

```
int main()
{
   car car1;
    ...
}
```

作成時に呼び出される

図 13-2 コンストラクタ
コンストラクタを定義しておくと、オブジェクトを作成するときに自動的にその処理が行われます。

重要

オブジェクトを初期化するために、コンストラクタを定義することができる。

クラスの設計についての注意

この章以降では、説明をかんたんにするため、第12章で説明したようなデータメンバに代入される値をチェックするようなメンバ関数は定義していません。実際にクラスを設計する際には、誤りのないプログラムを作成するために、さまざまな工夫をこらすようにします。

13.2 コンストラクタのオーバーロード

コンストラクタをオーバーロードする

コンストラクタはメンバ関数のひとつです。関数では、引数の数・型が異なっていれば、同じ名前の関数を複数定義することができました。このしくみを「オーバーロード」と呼んでいたことを思い出してください。

コンストラクタも、引数の型や数が異なっていれば、同じようにオーバーロードすることができます。つまり、**複数のコンストラクタを定義する**ことができるのです。

次の2つのコンストラクタをみてください。

```
//Carクラスメンバ関数の定義
Car::Car()
{
   num = 0;
   gas = 0.0;
   cout << "車を作成しました。¥n";
}
Car::Car(int n, double g)
{
   num = n;
   gas = g;
   cout << "ナンバー" << num << "ガソリン量" << gas <<
 "の車を作成しました。¥n";
}
```

引数のないコンストラクタです

引数を2個もつコンストラクタです

この2つのコンストラクタが使われるコードを作成してみましょう。

Sample2.cpp ▶ コンストラクタをオーバーロードする

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
  private:
     int num;
     double gas;
  public:
     Car();
     Car(int n, double g);
     void show();
};

//Carクラスメンバ関数の定義
Car::Car()
{
  num = 0;
  gas = 0.0;
  cout << "車を作成しました。¥n";
}
Car::Car(int n, double g)
{
  num = n;
  gas = g;
  cout << "ナンバー" << num << "ガソリン量" << gas <<
 "の車を作成しました。¥n";
}
void Car::show()
{
  cout << "車のナンバーは" << num << "です。¥n";
  cout << "ガソリン量は" << gas << "です。¥n";
}

int main()
{
  Car car1;
  Car car2(1234, 20.5);

  return 0;
}
```

引数のないコンストラクタです

引数を2個もつコンストラクタです

引数のないコンストラクタが呼び出されます

引数を2個もつコンストラクタが呼び出されます

Lesson 13

Sample2の実行画面

```
車を作成しました。
ナンバー1234ガソリン量20.5の車を作成しました。
```

引数のないコンストラクタからの出力です

引数を2個もつコンストラクタからの出力です

このコードでは、2つのオブジェクトを作成しています。1つ目のオブジェクトでは、これまでと同じように記述しています。引数を指定していません。

```
Car car1;
```

引数のないコンストラクタを呼び出します

一方、2つ目のオブジェクトでは、2つの引数を指定しています。

```
Car car2(1234, 20.5);
```

引数を2個もつコンストラクタを呼び出します

このとき、それぞれ次のコンストラクタが自動的に呼び出されることになります。

1つ目のオブジェクト ⟶ 引数なしのコンストラクタ
2つ目のオブジェクト ⟶ 引数2個のコンストラクタ

1台目はナンバーとガソリン量が0の車、2台目はナンバーが1234でガソリン量が20.5の車として作成することができました。

つまり、コンストラクタを複数定義しておけば、いろいろな引数を渡して、オブジェクトを柔軟に作成できるようになるというわけです。

なお、コンストラクタを呼び出す場合には、次のように表記することもできます。

```
Car car1 = Car();
Car car2 = Car(1234, 20.5);
```

引数のないコンストラクタを呼び出します

引数を2個もつコンストラクタを呼び出します

この方法はやや長い記述となります。Sample1のように短い表記を使う方法が一般的です。

重要

コンストラクタをオーバーロードすることができる。

```
...
Car::Car()
{
  ...
}
Car::Car(int n, double g);
{
  ...
}
...
```

```
int main()
{
   Car car1;

   Car car2(1234, 20.5);
}
```

引数なし
引数が2個

図13-3 コンストラクタのオーバーロード
コンストラクタをオーバーロードすると、引数によって、適切なコンストラクタが呼び出されます。

コンストラクタを省略すると?

では、コンストラクタを1つも定義しなかった場合は、どのような処理が行われるのでしょうか? 第12章を思い出してください。第12章ではコンストラクタを定義しないCarクラスを設計していました。

実は、コンストラクタが定義されない場合は、引数をとらないコンストラクタが用意されます。このコンストラクタは何の処理もしません。このコンストラクタを、**デフォルトコンストラクタ**(default constructor)といいます。つまり、コンストラクタを省略した場合には、

コンパイラが用意した空のデフォルトコンストラクタが呼び出される

ということになります。

ただし、コンストラクタを何かしら1つでも自分で定義した場合は、空のデフォルトコンストラクタは用意されませんので注意してください。

13.3 コンストラクタの応用

オブジェクトの配列を作成する

コンストラクタを定義したクラスについて、いろいろなコードを記述してみましょう。まず最初に、オブジェクトの配列を作成するコードを記述してみることにします。クラスは新しい型のひとつですから、配列を宣言することもできます。配列を宣言すると、オブジェクトをまとめて扱うことができるようになります。次のコードをみてください。

Sample3.cpp ▶ オブジェクトの配列を作成する

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
   public:
      Car();
      Car(int n, double g);
      void show();
};

//Carクラスメンバ関数の定義
Car::Car()
{
   num = 0;
   gas = 0.0;
   cout << "車を作成しました。¥n";
}
```

引数のないコンストラクタの定義です

```
Car::Car(int n, double g)
{
   num = n;
   gas = g;
   cout << "ナンバー" << num << "ガソリン量" << gas <<
 "の車を作成しました。¥n";
}
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
   cout << "ガソリン量は" << gas << "です。¥n";
}

int main()
{
   Car mycars[3]={
      Car(),
      Car(1234,25.5),
      Car(4567,52.2)
   };

   return 0;
}
```

引数を２個もつコンストラクタの定義です

引数のないコンストラクタを呼び出します

引数を２個もつコンストラクタを呼び出します

Sample3の実行画面

```
車を作成しました。
ナンバー1234ガソリン量25.5の車を作成しました。
ナンバー4567ガソリン量52.2の車を作成しました。
```


このコードではオブジェクトの配列を宣言するため、{}内に，で区切った初期化子を記述しています。{}を使った初期化は第9章と同じです。このコードでは、1番目のオブジェクトは引数なしのコンストラクタで初期化しています。2、3番目のオブジェクトは引数2つのコンストラクタで初期化しています。

このコードをみればわかるように、この方法では多くの要素をもつ配列の場合には、初期化の記述が面倒です。そこで、初期化子を記述しない場合のコードをみてみましょう。

Sample4.cpp ▶ デフォルトコンストラクタを用意する

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
   public:
      Car();
      Car(int n, double g);
      void show();
};

//Carクラスメンバ関数の定義
Car::Car()
{
   num = 0;
   gas = 0.0;
   cout << "車を作成しました。¥n";
}
Car::Car(int n, double g)
{
   num = n;
   gas = g;
   cout << "ナンバー" << num << "ガソリン量" << gas <<
 "の車を作成しました。¥n";
}
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
   cout << "ガソリン量は" << gas << "です。¥n";
}

int main()
{
   Car cars[3];

   return 0;
}
```

初期化子なしで配列を作成するには、引数のないコンストラクタが必要です

引数のないコンストラクタが呼び出されます

Sample4の実行画面

```
車を作成しました。
車を作成しました。
車を作成しました。
```

引数のないコンストラクタが呼び出されます

オブジェクトの配列は、初期化子なしで記述することもできます。ただし、このときには必ず引数のないコンストラクタが呼び出されることになります。このため、もし引数が2つ以上あるコンストラクタを定義した場合には、必ず引数のないコンストラクタもあわせて定義しなければなりません。コンストラクタを1つでも自分で定義した場合は、デフォルトコンストラクタが用意されないためです。

デフォルト引数を利用する

ところで、第7章で学んだデフォルト引数を使うと、コンストラクタをシンプルに記述できる場合もあります。次の例をみてください。

Sample5.cpp ▶ デフォルト引数をもつコンストラクタを利用する

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
   public:
      Car(int n=0, double g=0);
      void show();
};

//Carクラスメンバ関数の定義
Car::Car(int n, double g)
{
   num = n;
```

デフォルト引数をもつコンストラクタです

```
    gas = g;
    cout << "ナンバー" << num << "ガソリン量" << gas <<
 "の車を作成しました。¥n";
}
void Car::show()
{
    cout << "車のナンバーは" << num << "です。¥n";
    cout << "ガソリン量は" << gas << "です。¥n";
}

int main()
{
    Car car1;
    Car car2(1234, 20.5);

    return 0;
}
```

引数を渡さないでオブジェクトを作成します

引数を渡してオブジェクトを作成します

Sample5の実行画面

```
車を作成しました。
ナンバー1234ガソリン量20.5の車を作成しました。
```

このコードでは、引数を2つもつコンストラクタを1つだけ定義しています。このコンストラクタは引数を渡さずに呼び出すことも、渡して呼び出すこともできます。デフォルト引数を使うことによって複数のコンストラクタを1つにまとめ、シンプルにコードを記述しているのです。

13.4 静的メンバ

メンバとオブジェクト

この節では、さらに特殊なメンバについて学ぶことにしましょう。

車クラスのオブジェクトについてふりかえってみてください。オブジェクトを作成することによって、それぞれのオブジェクトのnumやgasに値を代入したり出力したりすることができましたね。車のナンバーやガソリンといった値がそれぞれの車に存在しているように、**各オブジェクトごとにデータメンバに値を格納することができた**のです。

このことを、データメンバnum・gasは

オブジェクトに関連づけられている

ということもあります。

またshow()メンバ関数も、オブジェクトを作成することによって、呼び出すことができました。このメンバ関数もオブジェクトに関連づけられています。このように、通常のメンバはオブジェクトに関連づけられているのです。

図 13-4 **通常のメンバ**
通常のメンバは、オブジェクトが作成されたときにアクセスすることができます。

静的メンバのしくみを知る

実はクラスでは、オブジェクトに関連づけられていないメンバをもつことができます。これを、

クラス全体に関連づけられている

ということがあります。クラスに関連づけられているメンバを**静的メンバ**といいます。

静的メンバとなるデータメンバ・メンバ関数は、宣言するときに**static**という記憶クラス指定子をつけることになっています。実際にコードを入力しながら学ぶことにしましょう。次のコードをみてください。

Sample6.cpp ▶ 静的メンバを記述する

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
   public:
      static int sum;
      Car();
      void setCar(int n, double g);
      void show();
      static void showSum();
};

//Carクラスメンバ関数の定義
Car::Car()
{
   num = 0;
   gas = 0.0;
   sum++;
   cout << "車を作成しました。¥n";
}
void Car::setCar(int n, double g)
```

静的データメンバです

静的メンバ関数です

コンストラクタが呼び出されたときに静的データメンバsumの値を1増やします

```
{
   num = n;
   gas = g;
   cout << "ナンバーを" << num << "にガソリン量を" << gas <<
 "にしました。¥n";
}
void Car::showSum()
{
   cout << "車は全部で" << sum << "台あります。¥n";
}
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
   cout << "ガソリン量は" << gas << "です。¥n";
}

int Car::sum = 0;

//Carクラスの利用
int main()
{
   Car::showSum();

   Car car1;
   car1.setCar(1234, 20.5);

   Car::showSum();

   Car car2;
   car2.setCar(4567, 30.5);

   Car::showSum();

   return 0;
}
```

静的メンバ関数の定義です

静的データメンバを初期化します

静的メンバ関数を呼び出します

オブジェクトを作成します

もう一度静的メンバ関数を呼び出します

Sample6の実行画面

```
車は全部で0台あります。
車を作成しました。
ナンバーを1234にガソリン量を20.5にしました。
車は全部で1台あります。
車を作成しました。
ナンバーを4567にガソリン量を30.5にしました。
```

静的メンバ関数からの出力は0台です

オブジェクトが作成されると・・・

静的メンバ関数からの出力は1台になります

```
車は全部で２台あります。
```

ここでは、データメンバsumにstaticをつけて静的メンバとしています。

```
static int sum;
```
staticをつけた静的データメンバです

このsumはクラスに関連づけられますから、それぞれのオブジェクトのコンストラクタ内では初期化しません。Car::をつけて関数の外で初期化します。そして、車オブジェクトが1つ作成されるたびに、コンストラクタの中で「sum++;」という文が処理され、1つずつ値が増やされるようにしています。つまりsumは、

クラス全体で何台の車（いくつのオブジェクト）が存在しているか

をあらわすデータメンバとなっているのです。このように**クラス全体で扱うデータを格納しておくデータメンバ**が静的データメンバなのです。

次に、staticをつけたメンバ関数であるshowSum()関数の宣言をみてください。

```
static void showSum();
```
staticをつけた静的メンバ関数です

静的メンバとなっているメンバ関数は、

そのクラスからオブジェクトが作成されていなくても呼び出すことができる

というしくみをもつメンバ関数です。通常のメンバのようにオブジェクトに関連づけられたメンバではありません。

静的メンバ関数は、静的データメンバを出力したり、クラス全体にかかわる処理を行うものです。静的メンバ関数は、オブジェクトが作成されていなくても呼び出せるように、次のように呼び出します。

構文　静的メンバ関数の呼び出し

```
クラス名::関数名(引数リスト);
```

上のコードでは、次のように静的メンバ関数を呼び出しています。

```
Car::showSum();
```

クラス名をつけて呼び出します

この静的メンバ関数では、静的データメンバsumの値を出力します。sumの値は最初は0ですが、オブジェクトを1つ作成したあとにもう一度呼び出すと、1になっていることがわかります。

静的メンバを使って、クラス全体で車が何台あるのかを管理することができました。

重要

クラスに関連づけられているメンバを静的メンバという。

Carクラス
2
sum
クラスに関連づけられている
＝静的メンバ
オブジェクトに関連づけられている
＝通常のメンバ
Carクラスの
オブジェクト
num
gas
num
gas

図13-5　静的メンバ
静的メンバは、クラスに関連づけられたメンバです。

静的メンバについての注意

静的メンバ関数内では、通常のメンバにアクセスすることができません。静的メンバ関数は、特定のオブジェクトに関連づけられたものではないからです。

静的メンバ関数は、オブジェクトが作成されていなくても呼び出される場合があります。そこで、特定のオブジェクトに関連づけられている通常のメンバにアクセスすることができないのです。

つまり、次のコードは誤りとなります。

```
void Car::showSum()
{
    //誤り
    //cout << "車のナンバーは" << num << "です。";
}
```

静的メンバ関数内では・・・

インスタンス変数にアクセスできません

13.5 レッスンのまとめ

この章では、次のようなことを学びました。

- コンストラクタは、オブジェクトを作成するときに呼び出されます。
- 引数の型・数の異なるコンストラクタをオーバーロードすることができます。
- 引数を渡さずに呼び出せるコンストラクタを、デフォルトコンストラクタと呼びます。
- クラスに関連づけられたメンバを、静的メンバと呼びます。
- 静的データメンバには、クラス全体で管理するデータを格納します。
- 静的メンバ関数は、オブジェクトが作成されていなくても呼び出すことができます。

クラスにはさまざまな機能があります。初期化処理を行うために、コンストラクタを定義することができます。また、クラス全体に関連づけられるデータを管理するために、静的メンバを定義することができます。これらはクラスには欠かせない機能となっています。

練習

1. 次の項目について○か×で答えてください。

① クラスには必ず１つ以上のコンストラクタを記述する必要がある。
② コンストラクタは戻り値をもたない。
③ コンストラクタは引数をとらない。

2. 次の項目について○か×で答えてください。

① 静的データメンバには、オブジェクトが作成されていないとアクセスできない。
② 静的でない通常のデータメンバには、オブジェクトが作成されていなくてもアクセスできる。
③ 静的メンバ関数内では、静的でない通常のデータメンバにアクセスできる。

Lesson 14

新しいクラス

第12章と第13章では、クラスのさまざまな機能を学んできました。C++では、すでに設計したクラスを使って、新しいクラスを効率よく作成することができます。既存のクラスを利用して、プログラムを効率的に作成していくことができるのです。この章では、新しいクラスの作成方法を学んでいきましょう。

Check Point!

- 継承
- 派生
- 基本クラス
- 派生クラス
- 仮想関数
- 純粋仮想関数
- 抽象クラス
- 多重継承
- 仮想基本クラス

14.1 継承

継承のしくみを知る

これまでの章では、「車」の機能をまとめたクラスを使って、プログラムを作成してきました。この章では、さらに新しいプログラムを作成していくことにしましょう。

今度はたとえば、

競技用のレーシングカー

を扱うプログラムを作成することを考えてみてください。レーシングカーは車の一種ですから、車とレーシングカーには多くの共通点があります。

C++では、すでに作成したクラスをもとにして、新しいクラスを作成することができるようになっています。これまでの車をあらわす「Carクラス」をもとにして、レーシングカーをあらわす「RacingCarクラス」を作成できるようになっているのです。

このように、新しいクラスを作成することを、

クラスを派生する（extends）

といいます。

新しいクラスは、既存のクラスのメンバを「受け継ぐ」しくみになっています。既存のクラスに新しく必要となる性質や機能（メンバ）をつけたすように、コードを書いていくことができるのです。

次の記述をみてください。新しいクラスを作成するということは、おおよそこのような感じになります。

```
class 車 {
    ナンバー;
    ガソリン量;
    ナンバーとガソリン量を表示する機能・・・
};

class レーシングカー : 車 {
    競技用コース;
    競技用コースを表示する機能・・・
};
```

もとになるクラスです

新しいクラスです

追加する性質と機能をまとめます

「車」クラスのあとに、「レーシングカー」クラスをまとめてみました。レーシングカークラスは車クラスのメンバを受け継ぎます。このため、車クラスにあるメンバをコード中にもう一度書く必要はありません。レーシングカー独自の機能だけを書けばよいのです。

このように、新しく拡張したクラスが既存のクラスのメンバを受け継ぐことを、**継承**（inheritance）と呼びます。このとき、もとになる既存のクラスは**基本クラス**（base class）、新しいクラスは**派生クラス**（class）と呼ばれます。つまりここでは、

「車」クラス → 基本クラス
「レーシングカー」クラス → 派生クラス

となっているわけです。

基本クラス

派生クラス

図14-1 クラスの派生
既存のクラス（基本クラス）から新しいクラス（派生クラス）を作成することができます。

Lesson 14

クラスを拡張する

それではコードを記述して、クラスを派生する方法をおぼえましょう。派生クラスの宣言をするには、:のあとに基本クラスの名前を指定します。

構文　派生クラスの宣言

```
class 派生クラス名 : アクセス指定子 基本クラス名
{
   派生クラスに追加するメンバの宣言
};
```

派生クラスは次のようになります。実際のコードをみてください。

Sample1.cpp 前半 ▶ クラスを拡張する

```
#include <iostream>
using namespace std;

//Carクラスの宣言                  ← 基本クラスの宣言です
class Car{
   private:
      int num;
      double gas;
   public:
      Car();
      void setCar(int n, double g);
      void show();
};

//RacingCarクラスの宣言
class RacingCar : public Car{     ← 派生クラスの宣言です
   private:
      int course;                 ← 追加するデータメンバです
   public:
      RacingCar();                ← 派生クラスのコンストラクタです
      void setCourse(int c);      ← 追加するメンバ関数です
};

//Carクラスメンバ関数の定義
```

```
Car::Car()
{
    num = 0;
    gas = 0.0;
    cout << "車を作成しました。¥n";
}
void Car::setCar(int n, double g)
{
    num = n;
    gas = g;
    cout << "ナンバーを" << num << "にガソリン量を" << gas <<
 "にしました。¥n";
}
void Car::show()
{
    cout << "車のナンバーは" << num << "です。¥n";
    cout << "ガソリン量は" << gas << "です。¥n";
}

//RacingCarクラスメンバ関数の定義
RacingCar::RacingCar()
{
    course = 0;
    cout << "レーシングカーを作成しました。¥n";
}
void RacingCar::setCourse(int c)
{
    course = c;
    cout << "コース番号を" << course << "にしました。¥n";
}
・・・(後半に続く)
```

基本クラスCarと派生クラスRacingCarの宣言を記述しました。RacingCarクラスはCarクラスのメンバを継承します。このため、RacingCarクラス内では、受け継いだメンバについて特に記述する必要はありません。Carクラスにはない、独自のメンバだけを書けばよいのです。ここではデータメンバcourseとメンバ関数setCourse()が独自のメンバとなっています。

Lesson 14

重要

基本クラスを派生して派生クラスを宣言することができる。
派生クラスは基本クラスのメンバを継承する。

図 14-2 Car クラスと RacingCar クラス
基本クラス Car を派生して派生クラス RacingCar を宣言することができます。

派生クラスのオブジェクトを作成する

それでは、Sample1の前半に続けて、派生クラスのオブジェクトを作成するコードを入力しましょう。オブジェクトを作成する方法は、これまでと同様に派生クラスの変数を宣言します。

Sample1.cpp 後半 ▶ 派生クラスのオブジェクトを作成する

```
・・・(前半から続く)
int main()
{
    RacingCar rccar1;
    rccar1.setCar(1234, 20.5);
    rccar1.setCourse(5);

    return 0;
}
```


Sample1 の実行画面

```
車を作成しました。
レーシングカーを作成しました。
ナンバーを1234にガソリン量を20.5にしました。
コース番号を5にしました。
```

Sample1では、オブジェクトを作成したあと、次のようにメンバ関数を呼び出していますね。

```
rccar1.setCar(1234, 20.5);
rccar1.setCourse(5);
```

① 継承したメンバ関数の呼び出しです
② 追加したメンバ関数の呼び出しです

setCar()メンバ関数（①）は基本クラスで定義されているメンバ関数です。このメンバはサブクラスに継承されていますので、これまでと同じように基本クラスのオブジェクトで呼び出すことができます。

また、基本クラスで新しく追加したsetCourse()メンバ関数（②）も同じように呼び出すことができます。

基本クラスでは、継承したメンバと追加したメンバをどちらも同じように呼び出すことができるのです。このようにクラスを拡張することで、すでに設計したクラスから効率よく新しいクラスを作成することができます。つまり、プログラムを効率的に作成できる、というわけです。

重要

基本クラスを拡張して派生クラスを設計することができる。

図14-3 クラスの派生
クラスを派生すると、効率よくプログラムを作成することができます。

クラスの機能

ここで説明した「継承」と、「カプセル化」・「多態性」という３つの機能は、クラスがもつ強みとなっています。クラスによって、誤りの少ないプログラグラグラムを効率的に作成していくことができるのです。

基本クラスのコンストラクタを呼び出す

さて、次のSample1の実行結果をよくみてください。

Sample1 の実行画面

```
車を作成しました。
レーシングカーを作成しました。
・・・
```


最初に「車を作成しました。」と出力されていることから、派生クラスのオブジェクトを作成したときには、基本クラスのコンストラクタの処理が先に行われていることがわかりますね。

このように特に何も指定しないと、派生クラスのオブジェクトが作成されたときには、

派生クラスのコンストラクタ内の先頭で、
基本クラスの引数のないコンストラクタを呼び出す

ということになっています。

基本クラスのコンストラクタは、派生クラスに継承されません。そのかわり、このように基本クラスの引数なしのコンストラクタが自動的に呼び出されることになっています。こうして、基本クラスから継承したメンバの初期化が無事に行われるしくみになっています。

```
Car::Car()
{

}

Car::Car(int n, double g)
{

}
```

```
RacingCar::RacingCar()
{

}
```

何も指定がないと、基本クラスの引数なしのコンストラクタが最初に呼び出される

図 14-4 コンストラクタに何も指定しない場合
派生クラスのコンストラクタ内の処理の先頭で、基本クラスの引数のないコンストラクタが呼び出されます。

基本クラスのコンストラクタを指定する

このように、コンストラクタに特に何も指定しないと、最初に基本クラスの「引数なしのコンストラクタ」が呼び出されることがわかりました。

しかし、基本クラスにコンストラクタが複数ある場合には、呼び出されるコンストラクタを明示的に指定したい場合もあります。

このとき、派生クラスのコンストラクタを、次のように定義します。

Sample2.cpp ▶ 基本クラスのコンストラクタ

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
   public:
      Car();
      Car(int n, double g);
      void setCar(int n, double g);
      void show();
```

```
};

//RacingCarクラスの宣言
class RacingCar : public Car{
   private:
      int course;
   public:
      RacingCar();
      RacingCar(int n, double g, int c);
      void setCourse(int c);
};

//Carクラスメンバ関数の定義
Car::Car()
{
   num = 0;
   gas = 0.0;
   cout << "車を作成しました。¥n";
}
Car::Car(int n, double g)
{
   num = n;
   gas = g;
   cout << "ナンバー" << num << "ガソリン量" << gas <<
 "の車を作成しました。¥n";
}
void Car::setCar(int n, double g)
{
   num = n;
   gas = g;
   cout << "ナンバーを" << num << "にガソリン量を" << gas <<
 "にしました。¥n";
}
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
   cout << "ガソリン量は" << gas << "です。¥n";
}

//RacingCarクラスメンバ関数の定義
RacingCar::RacingCar()
{
   course = 0;
   cout << "レーシングカーを作成しました。¥n";
}
```

```
RacingCar::RacingCar(int n, double g, int c) : Car(n, g)
{
   course = c;
   cout << "コース番号" << course << "のレーシングカーを作成しました。¥n";
}
void RacingCar::setCourse(int c)
{
   course = c;
   cout << "コース番号を" << course << "にしました。¥n";
}

int main()
{
   RacingCar rccar1(1234, 20.5, 5);

   return 0;
}
```

基本クラスの引数2個のコンストラクタが呼び出されるようにします

派生クラスの引数3個のコンストラクタが呼び出されるようにします

Sample2の実行画面

```
ナンバー1234ガソリン量20.5の車を作成しました。
コース番号5のレーシングカーを作成しました。
```

基本クラスの引数2個のコンストラクタの処理です

ここでは、派生クラスの引数3個のコンストラクタの先頭で次のように記述していますね。

基本クラスのコンストラクタを指定します

```
RacingCar::RacingCar(int n, double g, int c) : Car(n, g)
{
   course = c;
   cout << "レーシングカーを作成しました。¥n";
   cout << "コース番号" << course << "のレーシングカーを作成しました。¥n";
}
```

この派生クラスのコンストラクタには、:Car(n, g)と記述しています。これは、基本クラスのコンストラクタを指定したい場合に記述します。

構文　基本クラスのコンストラクタの指定

```
派生クラス名::派生クラスコンストラクタ(引数リスト):基本クラスコンストラクタ(引数リスト)
{
    派生クラスコンストラクタの本体定義
}
```

すると、今度は「引数のないコンストラクタ」ではなく、「引数2個のコンストラクタ」が最初に呼び出されていることがわかります。つまり、**基本クラスのどのコンストラクタを呼び出すかを自分で指定する**ことができるのです。

```
Car::Car()
{

}

Car::Car(int n, double g)
{

}
```

```
RacingCar::RacingCar(int n, double g, int c) : Car(int n, double g)
{

}
```


図14-5 基本クラスのコンストラクタを指定する
派生クラスのコンストラクタで、基本クラスのコンストラクタを指定することができます。

14.2 メンバへのアクセス

派生クラス内から基本クラスのメンバにアクセスする

第12章では、privateやpublicを指定して、メンバへのアクセスをコントロールする方法を学びました。このしくみによって、誤りのおきにくいプログラムを作成することができたわけです。

ここでは基本クラスと派生クラスという密接な関係にあるクラスについて、どのようなアクセスができるのかを学ぶことにします。ちょっと混みいっていますが、あわてずにひとつずつ学びましょう。

まず、**派生クラスから基本クラスのメンバにアクセスする場合**についてみてみましょう。

派生クラスは基本クラスのメンバを無制限に利用できるわけではありません。第12章では、privateメンバにはクラス外からアクセスすることができなかったことを思い出してください。基本クラスのprivateメンバは、派生クラス内からもアクセスすることはできないようになっています。

しかし、派生クラスと基本クラスは密接な関係があるため、このような制限が不便な場合もあります。

そこで、基本クラスのメンバの宣言で**protected**というアクセス指定子を使うことができます。Sample1のCarクラスのデータメンバを**protected**メンバとすると、RacingCarクラスからアクセスすることができます。

Lesson 14

```
//Carクラスの宣言
class Car{
   protected:
      int num;
      double gas;
   public:
      Car();
```

Carクラスのデータメンバをprotectedと指定します

```
        void setCar(int n, double g);
        void show();
};

//RacingCarクラスの宣言
class RacingCar : public Car{
    private:
        int course;
    public:
        RacingCar();
        void setCourse(int c);
        void newShow();
};

...
//RacingCarクラスメンバ関数の定義
...
void RacingCar::newShow()
{
    cout << "レーシングカーのナンバーは" << num << "です。¥n";
}
```

派生クラスからCarクラスのprotectedメンバにアクセスできます

基本クラスのprotectedメンバは、privateメンバと同様に外部からアクセスすることができません。しかし、privateメンバと異なり、派生クラスの内部からだけはアクセスすることができます。

```
private:                                              ×

protected:
    int num;                                          ○
    double gas;

public:
    void setCar();

class  RacingCar:public Car {

private:
    int course;

public:
    RacingCar(int c, int n, double g);
    void setCourse(int c);
    void newShow();
};

void RacingCar::newShow()
{
    cout << "レーシングカーのナンバーは" << num << "です。¥n";
}
```

図14-6 派生クラス内から基本クラス内のメンバへのアクセス
派生クラスの中から、基本クラスのメンバを利用するには基本クラスでpublicまたはprotectedが指定されていなければなりません。基本クラスのprivateメンバにはアクセスすることができません。

外部から派生クラスがもつ基本クラスのメンバにアクセスする

次に、**基本クラスや派生クラスの外から、派生クラスがもつ基本クラスのメンバにアクセスする場合**をみてみましょう。派生クラスに継承された基本クラスのメンバは、派生クラスのpublicメンバになるのでしょうか？　privateメンバになるのでしょうか？　これは、

派生クラスがどのように継承されているか

によって決まります。派生クラスの宣言部分である次のコードをみてください。

```
//RacingCarクラスの宣言
class RacingCar : public Car{
   ...
```

ここでは、基本クラスを「public」として継承しています。「publicな継承」とは次のことを意味しています。

- 基本クラスのpublicメンバを、派生クラスにおいてもpublicメンバとする
- 基本クラスのprotectedメンバを、派生クラスにおいてprotectedメンバとする
- 基本クラスのprivateメンバを、派生クラスにおいてprivateメンバとする

つまり、Carクラスのpublicなメンバ関数であるshow()、SetCar()関数は、派生クラスにおいてもpublicメンバとされ、派生クラスの外から呼び出すことができるようになっています。

これらのことを表14-1にまとめておきましょう。表14-1は、基本クラスのメンバが、どこからアクセスできるのかを示したものです。

表14-1：アクセス指定子

基本クラスでのアクセス指定	継承のしかた	派生クラスからの利用	外部からの利用
public	public	できる	できる
protected		できる	できない
private		できない	できない
public	protected	できる	できない（派生クラスのprotectedメンバとなるため）
protected		できる	できない
private		できない	できない
public	private	できる	できない（派生クラスのprivateメンバとなるため）
protected		できる	できない
private		できない	できない

```
private:                                        ×

protected:
   int num;
   double gas;                                  ×

public:
   void setCar(int n, double g);
   void show();                                 ○

class  RacingCar :public Car{

private:
   int course;

public:
   RacingCar(int c, int n, double g);
   void setCourse(int c);
   void newShow();
};
```

```
int main()
{
   RacingCar rccar1();

   //rccar1.num;

   rccar.show();

   return 0;
}
```

図 14-7 クラス外から派生クラスがもつ基本クラスのメンバへのアクセス
外部から派生クラスがもつ基本クラスのメンバを利用するには、そのメンバが基本クラスのpublicメンバであり、かつpublicとして派生クラスに継承されていなければなりません。

14.3 仮想関数

メンバ関数をオーバライドする

この章では、新しくメンバをつけ加えるように派生クラスを記述してきました。ところで、派生クラスで新しくメンバ関数を記述するときには、実は、

基本クラスとまったく同じ関数名・引数の数・型をもつメンバ関数を定義することができる

ようになっています。

たとえば、すでに設計したCarクラスにはshow()という名前のメンバ関数があります。このとき派生クラスであるRacingCarクラスにも、同じ関数名・引数の数・型をもつshow()関数を定義することができるのです。次のコードをみてください。

```
...
void Car::show()                   ● 基本クラスのshow()メンバ関数です
{
   cout << "車のナンバーは" << num << "です。¥n";
}
...
void RacingCar::show()             ● 派生クラスのshow()メンバ関数です
{
   cout << "レーシングカーのナンバーは" << num << "です。¥n";
}
...
```

2つのクラスの2つのメンバ関数は、まったく同じ引数の型、数、関数名をもっていますね。ところで、派生クラスは基本クラスのメンバを継承することを学びました。では、次のような使いかたをした場合は、どちらのshow()関数が呼び出

されることになるのでしょうか？　次のコードをみてください。

Sample3.cpp ▶ メンバ関数をオーバーライドする

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   protected:
      int num;
      double gas;
   public:
      Car();
      void setCar(int n, double g);
      void show();        基本クラスのshow()メンバ関数です
};

//RacingCarクラスの宣言
class RacingCar : public Car{
   private:
      int course;
   public:
      RacingCar();
      void setCourse(int c);
      void show();        派生クラスのshow()メンバ関数です
};

//Carクラスメンバ関数の定義
Car::Car()
{
   num = 0;
   gas = 0.0;
   cout << "車を作成しました。¥n";
}
void Car::setCar(int n, double g)
{
   num = n;
   gas = g;
   cout << "ナンバーを" << num << "にガソリン量を" << gas <<
 "にしました。¥n";
}
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
```

Lesson 14

```
    cout << "ガソリン量は" << gas << "です。¥n";
}

//RacingCarクラスメンバ関数の定義
RacingCar::RacingCar()
{
    course = 0;
    cout << "レーシングカーを作成しました。¥n";
}
void RacingCar::setCourse(int c)
{
    course = c;
    cout << "コース番号を" << course << "にしました。¥n";
}
void RacingCar::show()
{
    cout << "レーシングカーのナンバーは" << num << "です。¥n";
    cout << "ガソリン量は" << gas << "です。¥n";
    cout << "コース番号は" << course << "です。¥n";
}

int main()
{
    RacingCar rccar1;
    rccar1.setCar(1234, 20.5);
    rccar1.setCourse(5);

    rccar1.show();

    return 0;
}
```

「show()メンバ関数」を呼び出すと・・・

Sample3の実行画面

```
車を作成しました。
レーシングカーを作成しました。
ナンバーを1234にガソリン量を20.5にしました。
コース番号を5にしました。
レーシングカーのナンバーは1234です。
ガソリン量は20.5です。
コース番号は5です。
```

派生クラスのshow()メンバ関数が呼び出されています

派生クラスのオブジェクトを作成し、show()関数を呼び出してみました。すると、派生クラスのshow()関数が呼び出されていることがわかります。関数名・引数の数・型がまったく同じである場合は、派生クラスで新しく定義したほうの関数が呼び出されるのです。

このように、派生クラスで定義したメンバ関数が、基本クラスのメンバにかわって機能することを**オーバーライド**（overriding）といいます。

重要

派生クラスで定義したメンバ関数が、基本クラスのメンバにかわって機能することをオーバーライドという。

派生クラスのshow()関数を呼び出しているのだから、新しく定義した関数が呼び出されるのはあたりまえだと思う方もいるかもしれません。しかし、このことは重要ですので、しっかりおぼえておきましょう。

```
protected:
   int num;
   double gas;

public:
   void show();

class  RacingCar : public Car{

private:
   int course;

public:
   RacingCar(int n, double g, int c);
   void show();

};
```

```
int main
{
   RacingCar rccar1(350,1234,25.5);

   rccar1.show();

   return 0;
}
```

図14-8　オーバーライド
オーバーライドとは、派生クラスのメンバ関数が基本クラスのメンバ関数にかわって機能することです。

Lesson 14

基本クラスへのポインタを利用する

今度は、異なる状況で同じ形式のメンバ関数を呼び出したときのようすをみてみることにしましょう。その前にひとつ、おぼえておきたいことがあります。それは、

基本クラスのポインタを使うと、基本クラス自身だけでなく、派生クラスのオブジェクトをさすことができる

ということです。次のように、基本クラスのポインタには派生クラスのオブジェクトのアドレスを代入することができるのです。つまり、基本クラスのポインタを使って派生クラスのオブジェクトを扱うことができるのです。

```
Car* pCar;
RacingCar rccar1;
pCar = &rccar1;
```

基本クラスのポインタで派生クラスのオブジェクトをさすことができます

今度はこのように派生クラスのオブジェクトをさしている基本クラスのポインタを使って、show()関数を呼び出してみることにします。つまり、上のコードのあとに、

```
pCar->show();
```

派生クラスをさしている基本クラスのポインタを使ってshow()関数を呼び出します

という呼び出しを行うのです。

今度はどちらのクラスのshow()関数が呼び出されるのでしょうか？　次のコードを記述してみてください。CarクラスとRacingCarクラスの宣言はSample3と同じです。

Sample4.cpp ▶ 基本クラスへのポインタを利用する

```
...
int main()
{
```

```
    Car* pCars[2];                    ← 基本クラスのポインタを準備します

    Car car1;                         ← 基本クラスのオブジェクトを作成します
    RacingCar rccar1;                 ← 派生クラスのオブジェクトを作成します

    pCars[0] = &car1;                 ┐
    pCars[0]->setCar(1234, 20.5);     │ どちらも基本クラスへのポイン
                                      │ タの配列で扱うことができます
    pCars[1] = &rccar1;               ┘
    pCars[1]->setCar(4567, 30.5);

    for(int i=0; i<2 ;i++){
        pCars[i]->show();             ← show()メンバ関数を呼び出します
    }
    return 0;
}
```

Sample4 の実行画面

```
車を作成しました。
車を作成しました。
レーシングカーを作成しました。
ナンバーを1234にガソリン量を20.5にしました。
ナンバーを4567にガソリン量を30.5にしました。
車のナンバーは1234です。       ┐ 基本クラスのshow()メンバ関数が呼び出されます
ガソリン量は20.5です。         ┘
車のナンバーは4567です。       ┐ 基本クラスのshow()メンバ関数が呼び出されます
ガソリン量は30.5です。         ┘
```

このコードでは、CarクラスとRacingCarクラスのオブジェクトを1つずつ作成し、Carクラスへのポインタの配列（pCars[]）にそれぞれのアドレスを代入しました。しかし、このとき呼び出されているのは、どちらも基本クラスのshow()関数になっています。つまり、基本クラスのポインタが派生クラスのオブジェクトをさしていても、基本クラスのshow()関数が呼び出されてしまっているわけです。さきほどの例とは反対になっています。

仮想関数を定義する

基本クラスのポインタを使ってshow()関数を呼び出す場合には、基本クラスのshow()関数が呼び出されることがわかりました。

しかし、配列の要素のうち、pCars[1]はレーシングカーであるrccar1をさしているポインタなのですから、RacingCarクラスで新しく定義されたshow()関数が呼びだされたほうが直感的です。

C++においてこのように直感的な呼び出しが行われるようにするためには、基本クラスのメンバ関数を宣言する際に、**virtual**という指定をつけておく必要があります。

構文 **仮想関数の宣言**

```
virtual 基本クラスメンバ関数の宣言;
```

virtualをつけた関数のことを、**仮想関数**（virtual function）と呼びます。

実際のコードをみてみましょう。

Sample5.cpp ▶ 仮想関数のオーバーライドを利用する

```
#include <iostream>
using namespace std;

//Carクラスの宣言
class Car{
   protected:
      int num;
      double gas;
   public:
      Car();
      void setCar(int n, double g);
      virtual void show();
};

//RacingCarクラスの宣言
class RacingCar : public Car{
   private:
```

仮想関数とします

```
        int course;
    public:
        RacingCar();
        void setCourse(int c);
        void show();
};

//Carクラスメンバ関数の定義
Car::Car()
{
    num = 0;
    gas = 0.0;
    cout << "車を作成しました。¥n";
}
void Car::setCar(int n, double g)
{
    num = n;
    gas = g;
    cout << "ナンバーを" << num << "にガソリン量を" << gas <<
 "にしました。¥n";
}
void Car::show()
{
    cout << "車のナンバーは" << num << "です。¥n";
    cout << "ガソリン量は" << gas << "です。¥n";
}

//RacingCarクラスメンバ関数の定義
RacingCar::RacingCar()
{
    course = 0;
    cout << "レーシングカーを作成しました。¥n";
}
void RacingCar::setCourse(int c)
{
    course = c;
    cout << "コース番号を" << course << "にしました。¥n";
}
void RacingCar::show()
{
    cout << "レーシングカーのナンバーは" << num << "です。¥n";
    cout << "ガソリン量は" << gas << "です。¥n";
    cout << "コース番号は" << course << "です。¥n";
}
```

Lesson 14

```
int main()
{
   Car* pCars[2];

   Car car1;
   RacingCar rccar1;

   pCars[0] = &car1;
   pCars[0]->setCar(1234, 20.5);

   pCars[1] = &rccar1;
   pCars[1]->setCar(4567, 30.5);

   for(int i=0; i<2 ;i++){
      pCars[i]->show();   ● show()メンバ関数を呼び出すと・・・
   }

   return 0;
}
```

Sample5の実行画面

```
車を作成しました。
車を作成しました。
レーシングカーを作成しました。
ナンバーを1234にガソリン量を20.5にしました。
ナンバーを4567にガソリン量を30.5にしました。
車のナンバーは1234です。
ガソリン量は20.5です。
レーシングカーのナンバーは4567です。
ガソリン量は30.5です。
コース番号は0です。
```

基本クラスのshow()メンバ関数が呼び出されます

派生クラスのshow()メンバ関数が呼び出されます

このコードは基本クラスのメンバ関数にvirtualを指定しただけで、あとはSample4と同じです。

今度は、pCars[1]->show()の呼び出しによって、RacingCarクラスで新しく定義したshow()関数が呼び出されています。たしかに、新しく派生クラスで定義されたほうのメンバ関数が呼び出されていますね。

このようにメンバ関数を仮想関数としておくと、

ポインタがさしているオブジェクトの型に応じて、適切なメンバ関数が呼び出されるようになるのです。仮想関数のしくみによってオーバーライドが行われるのです。

オーバーライドとオーバーロード

ここで学んだ「オーバーライド」と似た用語として、「オーバーロード」があります。オーバーロードについては、第7章や第13章で学びました。オーバーロードは、**関数名が同じで、引数などの形式が異なる関数を定義すること**です。

一方、オーバーライドは、クラスを派生する際に**関数名・引数などがまったく同じである関数を定義すること**です。まぎらわしいので、この違いには気をつけてください。

14.4 抽象クラス

純粋仮想関数のしくみを知る

この節では、さらに進んだ概念を学ぶことにしましょう。次のようなかたちをしたメンバ関数を**純粋仮想関数**（pure virtual function）といいます。

構文　**純粋仮想関数**

```
virtual メンバ関数の宣言 = 0;
```

＝0と記述します

純粋仮想関数は仮想関数の宣言の最後に、**=0**という指定をつけたものです。純粋仮想関数は、処理の内容を定義しません。

そして、クラスの宣言内にこのような純粋仮想関数を1つでももつクラスは、

オブジェクトを作成することができない

ということになっています。このようなクラスを**抽象クラス**（abstract class）といいます。

たとえば、次のようなVehicleというクラスをみてください。

```
//Vehicleクラスの宣言
class Vehicle{
   protected:
      int speed;
   public:
      void setSpeed(int s);
      virtual void show() = 0;
};
```

純粋仮想関数です

このVehicleクラスは純粋仮想関数をもっています。Vehicleクラスではshow()

関数がどのような処理をするのかは定義しません。この純粋仮想関数を1つでももつ抽象クラスはオブジェクトを作成できません。つまり、次のようなコードを記述してオブジェクトを作成することはできません。

```
Vehicle vc;
```

抽象クラスの変数を宣言することはできません

```
Vehicle* pVc = new Vehicle;
```

動的にオブジェクトを作成することはできません

Vehicleクラス
=抽象クラス

オブジェクトを
作成することはできない

図14-9 抽象クラス
純粋仮想関数を1つ以上もつ抽象クラスからは、オブジェクトを作成することができません。

純粋仮想関数を利用する

抽象クラスは何のために必要なのでしょうか? 順を追ってみてみることにしましょう。

まず、抽象クラスVehicleからは、これまでと同じように派生クラスを派生させることができます。ただし抽象クラスの派生クラスのオブジェクトを作成できるようにするためには、

抽象クラスの純粋仮想関数の内容を派生クラスできちんと定義してオーバーライドする

という作業をしなければなりません。抽象クラスはオブジェクトを作成することができません。そして、純粋仮想関数の内容を派生クラスで定義しなければ、その派生クラスでもオブジェクトを作成できないことになっているのです。

それでは、抽象クラスを使ったコードをみてみましょう。

Sample6.cpp ▶ 抽象クラスを利用する

```
#include <iostream>
using namespace std;

//Vehicleクラスの宣言
class Vehicle{                          ← 抽象クラスです
   protected:
      int speed;
   public:
      void setSpeed(int s);
      virtual void show() = 0;          ← 純粋仮想関数show()です
};

//Carクラスの宣言
class Car : public Vehicle{             ← 抽象クラスから派生しました
   private:
      int num;
      double gas;
   public:
      Car(int n, double g);
      void show();                      ← show()メンバ関数をもちます
};

//Planeクラスの宣言
class Plane : public Vehicle{           ← 抽象クラスから派生しました
   private:
      int flight;
   public:
      Plane(int f);
      void show();                      ← show()メンバ関数をもちます
};

//Vehicleクラスメンバ関数の定義
void Vehicle::setSpeed(int s)
{
   speed = s;
   cout << "速度を" << speed << "にしました。¥n";
}
```

```
//Carクラスメンバ関数の定義
Car::Car(int n, double g)
{
   num = n;
   gas = g;
   cout << "ナンバー" << num << "ガソリン量" << gas <<
 "の車を作成しました。¥n";
}
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
   cout << "ガソリン量は" << gas << "です。¥n";
   cout << "速度は" << speed << "です。¥n";
}

//Planeクラスメンバ関数の定義
Plane::Plane(int f)
{
   flight = f;
   cout << "便" << flight << "の飛行機を作成しました。¥n";
}
void Plane::show()
{
   cout << "飛行機の便は" << flight << "です。¥n";
   cout << "速度は" << speed << "です。¥n";
}

int main()
{
   Vehicle* pVc[2];

   Car car1(1234, 20.5);
   pVc[0] = &car1;
   pVc[0]->setSpeed(60);

   Plane pln1(232);
   pVc[1] = &pln1;
   pVc[1]->setSpeed(500);

   for(int i=0; i<2 ;i++){
      pVc[i]->show();
   }
}
```

show()メンバ関数の処理を定義しました

show()メンバ関数の処理を定義しました

抽象クラスへのポインタの配列を準備します

1番目のオブジェクトはCarクラスです

2番目のオブジェクトはPlaneクラスです

show()メンバ関数を呼び出すと・・・

Sample6の実行画面

```
ナンバー1234ガソリン量20.5の車を作成しました。
速度を60にしました。
便232の飛行機を作成しました。
速度を500にしました。
車のナンバーは1234です。
ガソリン量は20.5です。
速度は60です。
飛行機の便は232です。
速度は500です。
```

オブジェクトのクラスに対応したshow()メンバ関数が呼び出されます

抽象クラスから2つの派生クラスを宣言しました。2つの派生クラスは、オブジェクトを作成できるように、それぞれのクラスに適したshow()メンバ関数の処理内容を定義しています。

さて、main()メンバ関数では、抽象クラスVehicleへのポインタの配列を準備しています。抽象クラスはオブジェクトを作成することはできませんが、そのクラスのポインタを準備して、派生クラスのオブジェクトをさすことができるようになっています。

抽象クラスの純粋仮想関数は必ず派生クラスでオーバーライドされますから、それぞれのオブジェクトのクラスに適したshow()メンバ関数が呼び出されていることがわかります。まとめてオブジェクトを操作することができていますね。

このようなことができるのは、

抽象クラスを拡張した派生クラスはどれも、
抽象クラスの純粋仮想関数（show()メンバ関数）と同じ名前のメソッドをもっている

からです。最初に説明したように、抽象クラスの派生クラスでは、必ず「show()」という名前のメンバ関数をもち、その処理内容が定義されていることになっているからです。

つまり抽象クラスを使えば、その派生クラスをまとめてかんたんに扱うことができます。抽象クラスによって、わかりやすいコードを記述することができるのです。

重要

抽象クラスの派生クラスでは、純粋仮想関数を定義する。
抽象クラスを使うと、わかりやすいコードを記述できる。

14.5 クラスの階層

クラスの階層を知る

これまで、基本クラスから派生クラスを派生するコードについてみてきました。C++では、派生クラスからさらにクラスを派生することができます。クラスを階層化することができるのです。

このとき、派生クラスに対して直接的に基本クラスとなっている基本クラス1を**直接基本クラス**（direct base class）、間接的に基本クラスとなっている基本クラス0を**間接基本クラス**（indirect base class）といいます。

基本クラス0

基本クラス1

派生クラス

図14-10 派生クラスから派生させる
C++では、派生クラスからさらにクラスを派生することができます。

このように派生を重ねることによってクラスの仕様を継承し、大きなプログラムを効率的に作成していけるようになっているのです。

多重継承のしくみを知る

クラスを派生するとき、2つ以上のクラスを継承した派生クラスを使いたい場合があります。このような継承を**多重継承**（multiple inheritance）といいます。

図14-11 多重継承
2つ以上の基本クラスから継承を行うことができます。

多重継承は、次のようにカンマで区切って、2つ以上の基本クラスを指定します。

構文 **多重継承**

```
class 派生クラス : アクセス指定子 基本クラス1, アクセス指定子 基本クラス2...{
...
};
```

2つの基本クラスから派生します

次のコードをみてください。

Sample7.cpp ▶ 多重継承を利用する

```
#include <iostream>
using namespace std;

//Base1クラスの宣言
class Base1{
   protected:
      int bs1;
   public:
      Base1(int b1=0){bs1=b1;}
      void showBs1();
};

//Base2クラスの宣言
class Base2{
   protected:
      int bs2;
   public:
```

基本クラスその1です

基本クラスその2です

```
        Base2(int b2=0){bs2=b2;}
        void showBs2();
};

//Derivedクラスの宣言
class Derived : public Base1, public Base2{
    protected:
        int dr;
    public:
        Derived(int d=0){dr=d;}
        void showDr();
};

//Base1クラスメンバ関数の定義
void Base1::showBs1()
{
    cout << "bs1は" << bs1 << "です。¥n";
}

//Base2クラスメンバ関数の定義
void Base2::showBs2()
{
    cout << "bs2は" << bs2 << "です。¥n";
}

//Derivedクラスメンバ関数の定義
void Derived::showDr()
{
    cout << "drは" << dr << "です。¥n";
}

int main()
{
    Derived drv;

    drv.showBs1();
    drv.showBs2();
    drv.showDr();

    return 0;
}
```

2つのクラスから派生しています

Sample7の実行画面

```
bs1は0です。
bs2は0です。
drは0です。
```

Derived派生クラスは、基本クラスBase1クラスとBase2クラスから派生されています。2つのクラスから多重継承を行っているのです。Base1クラスからshowBs1()関数、Base2クラスからshowBs2()関数を継承しています。

Base1クラスのメンバ

void showBs1()

Base2クラスのメンバ

void showBs2()

```
class Derived : public Base1, public Base2{
  ...
};
```

図14-12 多重継承によるメンバの継承
多重継承した派生クラスは、2つ以上の基本クラスのメンバを継承しています。

重要

2つ以上の基本クラスから多重継承することができる。

基本クラスが同じメンバ名をもつ場合

ところで、上の多重継承において、showBs1()関数とshowBs2()関数がどちらともshowBs()という関数名だったとしたらどうなるのでしょうか。このとき、「Derivedクラスのshow Bs()関数」を呼び出すコードを記述すると、どちらのクラ

スから継承したメンバであるかが判断できないために、コードをコンパイルすることができません。

```
drv.showBs(); ●── この呼び出しは誤りです
```

しかし、C++では、スコープ解決演算子::を使うことで、次のようにメンバを呼び出すコードを記述できるようになっています。

```
drv.Base1::showBs(); ●── Base1から継承したメンバを呼び出します
drv.Base2::showBs(); ●── Base2から継承したメンバを呼び出します
```

次のコードをみてみましょう。

Sample8.cpp ▶ 同じメンバ名をもつ基本クラスを利用する

```
#include <iostream>
using namespace std;

//Base1クラスの宣言
class Base1{
   protected:
      int bs1;
   public:
      Base1(int b1=0){bs1=b1;}
      void showBs();
};

//Base2クラスの宣言
class Base2{
   protected:
      int bs2;
   public:
      Base2(int b2=0){bs2=b2;}
      void showBs();
};

//Derivedクラスの宣言
class Derived : public Base1, public Base2{
```

2つの基本クラスに同じメンバ名があります

2つのクラスから派生しています

```
   protected:
      int dr;
   public:
      Derived(int d=0){dr=d;}
      void showDr();
};

//Base1クラスメンバ関数の定義
void Base1::showBs()
{
   cout << "bs1は" << bs1 << "です。¥n";
}

//Base2クラスメンバ関数の定義
void Base2::showBs()
{
   cout << "bs2は" << bs2 << "です。¥n";
}

//Derivedクラスメンバ関数の定義
void Derived::showDr()
{
   cout << "drは" << dr << "です。¥n";
}

int main()
{
   Derived drv;

   drv.Base1::showBs();
   drv.Base2::showBs();
   drv.showDr();

   return 0;
}
```


メンバ関数を呼び出すときに、「クラス名::」を指定すると、どちらから継承したshowBs()関数を呼び出せばよいのかというあいまいさを解決することができます。このため、Sample8は問題なくコンパイルすることができます。

ここでは、最初の呼び出しをBase1から継承したもの、2番目の呼び出しをBase2から継承したものとしています。このサンプルコードの実行結果はSample7と同じです。

```
Base1クラスのメンバ
  void showBs()
Base2クラスのメンバ
  void showBs()
class Derived : public Base1, public Base2{
};
```

```
int main
{
   Derived drv;
   drv.Bs1::showBs();
   drv.Bs2::showBs();
   return 0;
}
```

図14-13 多重継承でメンバ名が重なる場合
複数の基本クラスが同じメンバ名をもつ場合には、スコープ解決演算子を使ってあいまいさを解決します。

重要
多重継承した基本クラスのメンバが同じ名前のときは、スコープ解決演算子（::）を使って指定する。

仮想基本クラスのしくみを知る

多重継承を行うときには、メンバの呼び出しに注意しなければならないことが理解できたでしょうか。多重継承を行うときには、このほかにもメンバの呼び出しにあいまいさが生じる場合があります。図14-14をみてください。

多重継承が行われると、次のようなクラス階層ができる場合があります。

基本クラス0
基本クラス1　基本クラス2
派生クラス

図14-14 派生クラスが間接基本クラスを２つもつ場合
多重継承によって、派生クラスが間接基本クラスを２つもつ場合があります。

このとき派生クラスは、基本クラス1と基本クラス2をとおして、基本クラス0クラスのメンバを2つもつことになります。しかし、派生クラスのオブジェクトがもつ基本クラス0のメンバにアクセスするコードを記述すると、どちらのクラスをとおして継承されたメンバなのか判断することができません。このため、コードをコンパイルすることができなくなります。

そこで、図14-14のようなクラス階層をつくる場合には、基本クラス0を**virtual**という指定をして継承しておきます。すると、派生クラスはその中に基本クラス0をただ1つもつようになります。このような基本クラス0を**仮想基本クラス**（virtual base class）と呼びます。

構文 **仮想基本クラス**

```
class 派生クラス : アクセス指定子 virtual 基本クラス{
    ...
};
```

virtualを指定して基本クラスを仮想基本クラスとします

ただし、このvirtualは仮想関数で使ったvirtualとは意味が異なるので注意してください。次のコードは、Base0を仮想基本クラスとするコードです。

Sample9.cpp ▶ 仮想基本クラスとして継承する

```
#include <iostream>
using namespace std;
```

```
//Base0クラスの宣言
class Base0{
   protected:
      int bs0;
   public:
      Base0(int b0=0){bs0=b0;}
      void showBs0();
};

//Base1クラスの宣言
class Base1 : public virtual Base0{
   protected:
      int bs1;
   public:
      Base1(int b1=0){bs1=b1;}
      void showBs1();
};

//Base2クラスの宣言
class Base2 : public virtual Base0{
   protected:
      int bs2;
   public:
      Base2(int b2=0){bs2=b2;}
      void showBs2();
};

//Derivedクラスの宣言
class Derived : public Base1, public Base2{
   protected:
      int dr;
   public:
      Derived(int d=0){dr=d;}
      void showDr();
};

//Base0クラスメンバ関数の定義
void Base0::showBs0()
{
   cout << "bs0は" << bs0 << "です。¥n";
}

//Base1クラスメンバ関数の定義
void Base1::showBs1()
{
   cout << "bs1は" << bs1 << "です。¥n";
```

Base0を仮想基本クラスとして継承します

Base0を仮想基本クラスとして継承します

2つのクラスから派生しています

```
}

//Base2クラスメンバ関数の定義
void Base2::showBs2()
{
   cout << "bs2は" << bs2 << "です。¥n";
}

//Derivedクラスメンバ関数の定義
void Derived::showDr()
{
   cout << "drは" << dr << "です。¥n";
}

int main()
{
   Derived drv;

   drv.showBs0();

   return 0;
}
```

仮想基本クラスから継承された
メンバを呼び出すことができます

Sample9の実行画面

```
bs0は0です。
```

このコードでは、Derivedクラスの中にBase0クラスのメンバが1つ存在するようになっています。Base0クラスを仮想基本クラスとしているからです。このため、Base0クラスから継承したメンバを呼び出すことができます。もし仮想基本クラスとしなかった場合には、呼び出しはエラーとなります。

このように、多重継承を行うときには、あいまいさに注意しなければならないことがあります。

Lesson 14

重要

仮想基本クラスを作ることができる。

14.6 レッスンのまとめ

この章では、次のようなことを学びました。

- 基本クラスから派生クラスを派生させることができます。
- 派生クラスは、基本クラスのメンバを継承します。
- 基本クラスのprotectedメンバに、派生クラスからアクセスすることができます。
- 基本クラスと同じ関数名、引数の型・数をもつ関数を派生クラスで定義してオーバーライドすることができます。
- 基本クラスへのポインタを使って派生クラスのオブジェクトを扱う場合、virtualを指定して仮想関数とした関数だけがオーバーライドされます。
- 純粋仮想関数を１つ以上もつクラスを、抽象クラスと呼びます。
- 抽象クラスのオブジェクトを作成することはできません。
- ２つ以上の基本クラスから多重継承を行うことができます。
- １つの派生クラスに基本クラスが重複して継承される場合は、基本クラスを仮想基本クラスとしておくことができます。

この章では、既存のクラスから、新しいクラスを作成する方法を学びました。すでに設計したクラスを継承すると、より効率的にプログラムを作成していくことができます。既存のコードにつけたすように新しいコードを記述していくことができるからです。継承は、「クラス」の強力な機能のひとつとなっています。

練習

1. 次の項目について○か×で答えてください。

① 基本クラスから派生できるクラスの数は決まっている。

② 派生クラスからさらにクラスを派生させることを多重継承という。

③ 派生クラスでは、基本クラスと同じ名前のメンバ関数をもつことができる。

2. 次の項目について○か×で答えてください。

① 抽象クラスへのポインタは宣言することができない。

② 抽象クラスのオブジェクトは作成することができない。

③ 派生クラスのオブジェクトのアドレスは、基本クラスのポインタに代入できる。

3. 次のコードの誤りを指摘してください。

```
//Base1クラスの宣言
class Base1{
   protected:
      int bs1;
   public:
      Base1(int b1=0){};
      void showBs();
};

//Base2クラスの宣言
class Base2{
   protected:
      int bs2;
   public:
      Base2(int b2=0){};
      void showBs();
};

//Base1クラスメンバ関数の定義
void Base1::showBs()
```

```
{
   cout << "bs1は" << b1 << "です。¥n";
}

//Base2クラスメンバ関数の定義
void Base2::showBs()
{
   cout << "bs2は" << b2 << "です。¥n";
}

//Derivedクラスの宣言
class Derived : public Base1, public Base2{
   protected:
      int dr;
   public:
      Derived(int dr=0){}
      void showDr();
};

//Derivedクラスメンバ関数の定義
void Derived::showDr()
{
   cout << "drは" << dr << "です。¥n";
}

int main()
{
   Derived drv;
   drv.showBs();
   drv.showDr();

   return 0;
}
```

Lesson 15

クラスに関する高度なトピック

これまでの章で、クラスがもつさまざまな便利な機能について学びました。この章では、クラスがもつそのほかの高度な機能について学んでいきましょう。

Check Point!

- 演算子のオーバーロード
- 変換関数
- 変換コンストラクタ
- デストラクタ
- コピーコンストラクタ
- 代入演算子
- クラステンプレート
- 例外処理

15.1 演算子のオーバーロード

演算子のオーバーロードのしくみを知る

この章では、クラスに関するさまざまな高度なトピックを紹介していくことにしましょう。まず、クラスに対して演算子を使う方法を学ぶことにしましょう。

演算子とはどういうものなのか、ちょっと思い出してみてください。第4章では、+演算子や−演算子によってさまざまな演算を行いました。int型やdouble型などの基本型に対して演算を行ったわけです。

クラスは私たちが作成した新しい型です。オブジェクトに対しても、このような演算子を使えれば便利です。

まず、次のクラスをみてください。

座標をあらわす
Pointクラスです

```
//Pointクラスの宣言
class Point{
   private:
      int x;
      int y;
   public:
      Point(int a=0, int b=0){x=a; y=b;}
      void show(){cout<< "x :" << x << " y :" << y << '¥n';}
      setX(int a){x=a;}
      setY(int b){x=b;}
};
```

このクラスは、xy座標上の点をあらわすPointというクラスです。数学的なモノの概念を扱うために宣言したクラスです。このクラスを利用すれば、座標上の点をあらわすオブジェクトを作成することができます。

```
//Pointクラスの利用
int main()
{
   Point p1(1, 3);
   Point p2(5, 2);
   ...
}
```


さて、この2つの座標位置を「たしあわせた」位置をもつオブジェクトを、2つのオブジェクトの値から記述したい場合、どうしたらよいでしょうか？

このような数学的な概念をあらわすクラスでは、+演算子を使って次の「たし算」をすることができれば、わかりやすく、便利です。


```
//Pointクラスの利用
int main()
{
   Point p1(1, 3);
   Point p2(5, 2);
```

```
    Point p3 = p1 + p2;
    ...
}
```

このような演算が行えれば便利です

しかし、原則としてこのようなPointクラスのたし算はできません。+演算子は、私たちが作成したPointクラスのオブジェクトを扱う用法をもっていないからです。

そこでC++では、オブジェクトを扱う演算子を新しく定義することができるようになっています。つまり、

演算子について新しい用法を定義する

ことができるのです。これを、**演算子のオーバーロード**（operator overloading）といいます。

重要

C++では演算子の用法を定義することができる。

メンバ関数としてオーバーロードする

演算子をオーバーロードしてみましょう。演算子の用法は、一種の関数として定義します。演算子は

operator演算子記号()

というかわったかたちの関数として定義することになります。

たとえば、

```
p1+p2
```

という演算を行うことを考えてみてください。この+演算子を利用することを、次のようなメンバ関数の呼び出しであると考えるのです。

```
p1 .operator+ (p2);
```

＋演算子を「operator+()」という名のメンバ関数の呼び出しと考えます

つまり、p1というオブジェクトの「operator+()」というメンバ関数を呼び出すことなのだ、と考えるのです。したがって、Pointクラスにoperator+()メンバ関数を定義することで、+演算子を使えるようにするのです。この関数を、**演算子関数**(operator function) ともいいます。Pointクラスを加算する+演算子は、

Pointクラス型の引数を1つとり、Pointクラス型の値を戻すoperator+()という名の演算子関数

として定義することになります。

p1 / Pointクラス　+ / 演算子　p2 / Pointクラス

↓　↓　↓

p1 / Pointクラス　.operator+ / 演算子関数　(p2) / Pointクラスの引数

図15-1 演算子のオーバーロード
+演算子をオーバーロードしてオブジェクトを扱う新しい用法を定義することができます。

構文　2項演算子をメンバ関数としてオーバーロード

```
戻り値の型 operator演算子(引数1);
```

では実際に+演算子の新しい用法として、演算子関数を定義してみましょう。

```
//Pointクラスの宣言
class Point{
   private:
      int x;
      int y;
   public:
      Point(int a=0, int b=0){x=a; y=b;}
```

```
    Point operator+(Point p);
};

//Point クラスメンバ関数の定義
Point Point::operator+(Point p)
{
   Point tmp;
   tmp.x = x + p.x;
   tmp.y = y + p.y;
   return tmp;
}
```

+演算子関数の宣言をします
Pointクラスを扱う +演算子の処理内容です
左オペランドに対応します
右オペランドに対応します
演算結果です

+演算子について新しい用法を定義しました。Point型の引数を受けとって、そのメンバをデータメンバxとyに加算します。そしてその結果をPoint型の値として戻すように定義しているのです。

このような演算子関数を定義すると、Pointクラスのオブジェクトに対し、+演算子が利用できるようになります。次のように+演算子を使ったときに、定義した演算子関数が呼び出されるのです。

```
//Point クラスの利用
int main()
{
   Point p1(1, 2);
   Point p2(3, 6);
   p1 = p1 + p2;
    ...
}
```

+演算子が使えるようになります

+演算子が使えるようになれば、Pointクラスをより利用しやすくなりますね。

フレンド関数としてオーバーロードする

ところで、次のように+演算子の左オペランドとして整数値を使いたい場合を考えてみてください。さきほどの演算子関数はうまく機能するのでしょうか。

```
3+p1;
```


ただし、この演算は次のような座標を求めるものだとします。

実は、このように+演算子を使おうとしても、さきほど定義した演算子関数は呼び出されません。どうしてでしょうか？ それは、さきほどメンバ関数として定義した+演算子の左オペランドは、必ずPointクラスのオブジェクトでなければならないからです。

```
Point operator+(Point p);
```


図15-2 演算子をメンバ関数としてオーバーロードできない場合
2項演算子の左オペランドがそのクラスの値でない場合は、メンバ関数として定義した演算子は使えません。

そこで、演算子の左オペランドがそのクラスの以外の値を使う場合には、演算子を次のような関数としてオーバーロードすることになります。このfriendをつけた関数を**フレンド関数**（friend function）といいます。

構文　2項演算子をフレンド関数としてオーバーロード

```
friend 戻り値の型 operator演算子(引数1, 引数2);
```

ではさっそく、左オペランドがPointクラスでない場合に呼び出される演算子関数を定義してみましょう。

```
//Pointクラスの宣言
class Point{
   private:
      int x;
      int y;
   public:
      ...
      friend Point operator+(int a, Point p);
};
...
//フレンド関数の定義
Point operator+(int a, Point p)
{
   Point tmp;
   tmp.x = a + p.x;
   tmp.y = a + p.y;
   return tmp;
}
...
```


フレンド関数は、そのクラスのメンバ関数ではありません。しかし、フレンド関数は、

特別にそのクラスのprivateメンバにアクセスできる関数

となります。この演算子関数では、privateメンバを扱っています。そこで、フレンド関数としているのです。一般的に、メンバ関数とできない演算子関数は、フ

レンド関数とすることになります。

フレンド関数はクラス内に宣言しますが、クラスのメンバ関数ではありません。したがって、関数本体の定義にはPoint::をつけません。

重要

メンバ関数として定義できない演算子関数は、フレンド関数として定義する。

3　+　p2
int型　演算子　Pointクラス

operator+　(int a　,　Point p)
演算子関数　int型　Pointクラス

図15-3　**フレンド関数としてオーバーロード**
メンバ関数としてオーバーロードできない場合は、フレンド関数としてオーバーロードします。

フレンド関数

ここでは演算子関数をフレンド関数としました。このほか、通常の関数もフレンド関数として定義することができます。クラス内のprivateメンバにアクセスする関数として、フレンド関数を定義することができるのです。

ただし第12章でも説明したように、クラスでは、基本的にはprivateメンバにアクセスしないようにします。privateメンバに勝手にアクセスできる関数が増えると、カプセル化の意味がなくなってしまうからです。したがって、フレンド関数を定義するかどうかは十分に検討する必要があります。

メンバ関数・フレンド関数としてオーバーロードする

メンバ関数としてオーバーロードした演算子は、フレンド関数としてオーバーロードすることもできます。たとえば、最初に定義したp1+p2を行う+演算子は、メンバ関数とフレンド関数のどちらとしてもオーバーロードすることができるのです。2とおりの方法で、オーバーロードする方法をみてみましょう。

メンバ関数とする場合

```
class Point{
    ...
    Point operator+(Point p);
};

Point Point::operator+(Point p)
{
   Point tmp;
   tmp.x = x + p.x;
   tmp.y = y + p.y;
   return tmp;
}
```

引数を1つもつメンバ関数です

右オペランドに対応します

左オペランドに対応します

フレンド関数とする場合

```
class Point{
    ...
    friend Point operator+(Point p1, Point p2);
};

Point operator+(Point p1, Point p2)
{
   Point tmp;
   tmp.x = p1.x + p2.x;
   tmp.y = p1.y + p2.y;
   return tmp;
}
```

引数を2つもつフレンド関数です

右オペランドに対応します

左オペランドに対応します

上のコードは、どちらも同じ+演算子関数の内容を定義したものです。ただし、メンバ関数とフレンド関数では、引数の数が異なっていることに注意してください。

p1 + p2
Pointクラス 演算子 Pointクラス

operator+ (Point p1, Point p2)
演算子関数 Pointクラスの引数

図15-4 フレンド関数としてオーバーロード
メンバ関数とした演算子関数は、フレンド関数としてオーバーロードすることもできます。

演算子のオーバーロードについての注意

ここでは、+演算子をオーバーロードする方法についてみてきました。このほかの演算子もオーバーロードすることができます。演算子をオーバーロードするときの注意についてみておきましょう。

- 次の5つの演算子はオーバーロードできません。
  . :: .* ?: sizeof
- 第4章の表以外の新しい記号を演算子として定義することはできません。
- 基本型に対する演算子の用法をオーバーロードして変更することはできません。
- 演算子がとるオペランドの数は変更することができません。単項演算子であるものを2項演算子として定義したり、その逆にしたりすることはできません。
- 演算子の優先順位を変更することはできません。
- 演算子にデフォルト引数を定義することはできません。

上のような制限を守れば、さまざまな演算子をオーバーロードすることが可能です。ただし、その演算子がもつ意味と異なる処理をすることはするべきではないでしょう。たとえば、+演算子の処理を定義するならば、なんらかの「たし算」を連想させる処理を行うように定義するべきです。演算子関数にはさまざまな処

理を記述することができてしまいます。しかし、+演算子にたし算以外の操作を行わせる演算子関数は誤りのもととなります。

代入演算子

オブジェクトに対して演算子を使うためには、演算子をオーバーロードしなければなりません。ただし、代入演算子 (=) はオーバーロードしなくても使うことができます。代入演算子は、オブジェクトのメンバの値をもう一方のオブジェクトのメンバにコピーする機能をもちます。

ただし、もし代入演算子について規定の用法が適切でない場合には、自分で定義することができます。operator=()という演算子関数を定義すれば、定義した処理が行われるのです。代入演算子のオーバーロードは、この章の最後でとりあげることにします。

単項演算子をオーバーロードする

それでは、+演算子以外のオーバーロードを行ってみることにしましょう。

+演算子はオペランドを2つとる2項演算子でした。これに対し、++演算子のようにオペランドを1つだけとる単項演算子は、通常、**引数をとらないメンバ関数**として定義します。

構文　単項演算子をメンバ関数としてオーバーロード

```
戻り値の型 operator演算子();
```

単項演算子をメンバ関数としてオーバーロードする場合には引数をとりません

たとえば、次のように記述することで、x座標とy座標がともに1ずつインクリメントされるような++演算子を定義してみましょう。

```
++p1;
```

この++演算子関数の定義は次のようになります。

```
//Pointクラスメンバ関数の定義
Point Point::operator++()
{
   x++;
   y++;
   return *this;
}
```


ここでは、戻り値として*thisを指定しています。thisとは、**そのメンバ関数を呼び出した、オブジェクト自身へのポインタ**を意味しています。つまり、++p1;の場合はp1へのポインタを意味します。メンバ関数の中では、このthisというポインタを利用して、そのオブジェクト自身に関する情報を扱うことができます。

この関数内では、各メンバの値を1つ増やしたあとの自分自身を、戻り値としています。これは、自分自身を演算結果とすることで、インクリメントした「あと」の値を、ほかのオブジェクトに代入できるようにするためです。

```
Point p3= ++p1;
```

インクリメントしたあとの値を代入する様子については、第4章に戻って復習してみてください。

重要

thisポインタは、そのメンバ関数を呼び出したオブジェクト自身をさす。

図15-5　単項演算子のオーバーロード
単項演算子は引数をとらないメンバ関数として定義します。

ところで、インクリメント演算子は、前置と後置の2種類がありました(第4章)。前置インクリメント演算子は引数なしのメンバ関数として定義しましたが、後置インクリメント演算子は、前置と区別するため、引数が1つの演算子関数としてオーバーロードします。

```
Point Point::operator++(int d)
{
    Point p = *this;
    x++;
    y++;
    return p;
}
```


この後置インクリメント演算子は、thisを使って、pにインクリメント前の自分自身をいったん格納しておきます。最後にインクリメント前の値を演算結果として戻しているのです。これはインクリメントする「前」の値をほかのオブジェクトに代入するためです。int型の引数dは、前置の演算子と区別をつけるためだけに使います。

いろいろな演算子をオーバーロードする

ここまで、Pointクラスを扱うための演算子をオーバーロードしてきました。ここでまとめて1つのコードを記述してみましょう。

Sample1.cpp ▶ いろいろな演算子をオーバーロードする

```
#include <iostream>
using namespace std;

//Pointクラスの宣言
class Point{
    private:
        int x;
        int y;
```

```
    public:
        Point(int a = 0, int b = 0){x = a; y = b;}
        void setX(int a){x = a;}
        void setY(int b){x = b;}
        void show(){cout<< " x :" << x << " y :" << y
            << '¥n';}
        Point operator++();
        Point operator++(int d);
        friend Point operator+(Point p1, Point p2);
        friend Point operator+(Point p, int a);
        friend Point operator+(int a, Point p);
};

//Pointクラスメンバ関数の定義
Point Point::operator++()
{
    x++;
    y++;
    return *this;
}
Point Point::operator++(int d)
{
    Point p = *this;
    x++;
    y++;
    return p;
}

//フレンド関数の定義
Point operator+(Point p1, Point p2)
{
    Point tmp;
    tmp.x = p1.x + p2.x;
    tmp.y = p1.y + p2.y;
    return tmp;
}
Point operator+(Point p, int a)
{
    Point tmp;
    tmp.x = p.x + a;
    tmp.y = p.y + a;
    return tmp;
}
```

いろいろな演算子をオーバーロードします

前置インクリメント演算子の定義です

後置インクリメント演算子の定義です

p1+p2を行う+演算子の定義です

p1+3を行う+演算子の定義です

```
Point operator+(int a, Point p)
{
   Point tmp;
   tmp.x = a + p.x;
   tmp.y = a + p.y;
   return tmp;
}

int main()
{
   Point p1(1, 2);
   Point p2(3, 6);
   p1 = p1+p2;
   p1++;
   p1 = p1+3;
   p2 = 3+p2;

   p1.show();
   p2.show();

   return 0;
}
```

3+p1を行う＋演算子の定義です

定義した演算子を使っています

Sample1の実行画面

```
x:8 y:12
x:6 y:9
```

このコードでは、これまでに紹介した5種類の演算子関数を定義しています。ややこしかったかもしれませんが、ひとつずつ復習してみましょう。演算子をオーバーロードすることによって、クラスにより強力な機能をもたせることができるのです。

参照引数を使った演算子のオーバーロード

演算子関数の引数には、オブジェクトへの参照を使うこともできます。第12章で説明したように、参照を引数として使うと、処理速度が向上する場合があります。

たとえば、＋演算子関数に参照引数を使うと次のようになります。

```
Point operator+(const Point& p);
```

参照を引数として使っています

関数において実行速度をあげるという点ではオブジェクトへのポインタを使う方法も考えられます。しかし、＋演算子の場合は、形式上参照しか使うことができません。＋演算子のオペランドをポインタとすることはできないようになっているのです。

15.2 クラスの型変換

変換関数を利用する

演算子のオーバーロードについて理解できたでしょうか。次に、クラスの型変換を行う方法をみていきましょう。

まず、第4章で学んだ型変換を思い出してみてください。第4章では代入や演算の際に型変換が行われることを示しました。また、テキスト演算子を使って型変換を行うことも学びました。

クラスも新しい型のひとつです。異なる型どうしで型変換が行えると便利な場合があります。次のクラスをみてください。

```
//Numberクラスの宣言
class Number {
   private:
      int num;
   public:
      Number(){num = 0;}
      Number(int n){num = n;}          ← 変換コンストラクタです
      operator int(){return num;}      ← 変換関数の定義です
      Number operator++();
      Number operator++(int d);
      Number operator--();
      Number operator--(int d);
};

Number Number::operator++()
{
   num++;
   return *this;
}
Number Number::operator++(int d)
{
   Number n = *this;
   num++;
```

```
    return n;
}
Number Number::operator--()
{
    num--;
    return *this;
}
Number Number::operator--(int d)
{
    Number n = *this;
    num--;
    return n;
}
```

このコードは数をカウントするためのNumberというクラスを宣言したものです。Numberクラスの値は、int型の値との間で型変換を行うことができるように定義されています。

まず、Numberクラス中の次の部分をみてください。

```
operator int(){return num;}
```

変換関数の定義です

このようなかたちをしたメンバを、**変換関数**（conversion function）と呼びます。変換関数は、

クラスの値を、別の型の値に変換する

という役割をもっています。ここでは、Number型の値をint型の値に変換する処理を定義しているのです。

構文 **変換関数**

```
operator 型名()
```

変換関数は戻り値の型を指定しません。戻される値の型は、変換関数の型名であることがあきらかだからです。

このような変換関数を定義しておくと、キャスト演算子を使って明示的に型変換ができます。Numberクラスを利用している次のコードをみてください。

```
//Numberクラスの利用
int main()
{
   Number n;
   int i = (int)n;
   ...
```

Number型の値nをint型に変換できます

このNumberクラスの値は、キャスト演算子を使って、上のようにint型の値に変換することができるようになっています。

また、この変換関数はキャスト演算子を使わなくても機能します。したがって、上のコードは次のように書くこともできます。

```
int main()
{
   Number n;
   int i = n;
   ...
```

キャスト演算子を使わなくても、変換関数が定義されていれば型変換が行われます

キャスト演算子を明示的に使わなくてもint型の変数に代入できるのです。

重要

変換関数を定義すると、ほかのクラスへと型変換できる。

変換コンストラクタを利用する

このように、変換関数を定義することで、クラスの値を別の型の値に変換することができます。それでは、int型の値をNumberクラスの値に変換するにはどうしたらよいのでしょうか。Numberクラスの定義をみてください。ここに、引数が1つであるコンストラクタがあります。

```
Number(int n){num=n;}
```

int型の値をNumber型の値に変換するコンストラクタです

このようなコンストラクタを定義しておけば、オブジェクトを作成するときに、int型の値を1つ引数として渡すことで、Number型の値が得られます。

```
int i = 10;
Number n(i);
```

int型の値10をNumber型の値nに変換していると考えられます

つまり、引数1つのコンストラクタは変換関数の逆の機能をもっているのです。これを変換コンストラクタと呼びます。

構文 変換コンストラクタ

```
クラス名::クラス名(引数)
```

変換コンストラクタが定義してある場合にも、キャスト演算子を使わずにint型からNumber型への型変換が行われます。

```
int main()
{
   int i = 10;
   Number n = i;
...
```

重要 引数が1つのコンストラクタ(変換コンストラクタ)を定義すると、ほかのクラスから型変換できる。

Number(int n)

Number型の値 ⇄ int型の値

operator int()

図15-6 型変換
変換関数はクラス型の値を別の型の値に変換します。変換コンストラクタは別の型の値をクラス型の値に変換します。

15.3 メモリの確保と解放

デストラクタを定義する

この節では、クラスを設計するときに注意しなければならないことを学ぶことにしましょう。特に、クラス内で動的にメモリを確保するときには、さまざまな注意が必要になります。第9章では、値を記憶するために、new演算子を使って動的にメモリを確保する方法を学びました。クラスの内部においても、そのようなメモリの確保をする場合があります。次のコードをみてください。

```
#include <iostream>
#include <string>
using namespace std;

//Carクラスの宣言
class Car {
   private:
      int num;
      double gas;
      char* pName;
   public:
      Car(char* pN, int n, double g);
      ...
};

Car::Car(char* pN, int n, double g)
{
   cout << pN << "を作成します。¥n";
   pName = new char[strlen(pN)+1];
   strcpy(pName, pN);
   num = 0;
   gas = 0.0;
}
...
```

コンストラクタ内で動的にメモリを確保しています

このCarクラスは、車の名前をあらわす文字列を扱うのために、pNameというポインタをメンバとしてもっています。そして、Carクラスのコンストラクタ内では、車の名前をあらわすために、

文字列にあった大きさのメモリを確保する

という処理を行っています。したがって、このCarクラスのオブジェクトを作成すると、動的にメモリが確保され、pNameに文字列の先頭アドレスが格納されるわけです。

さて、このコードではnew演算子を使ってメモリを確保しているので、必ずコードのどこかでdelete演算子を記述し、メモリを解放しなければなりません（第10章）。このようにオブジェクトがメモリを動的に確保する処理を行う場合には、オブジェクトが破棄される前に確保したメモリを解放する必要があります。そうしないと、オブジェクトが作成されるたびに利用できるメモリが減っていってしまい、プログラムがうまく動作しなくなってしまうからです。

それではいったい、どこでこのメモリを解放すればよいのでしょうか？ これはオブジェクトが破棄されるとき、自動的に呼び出されるメンバ関数の中で定義します。このメンバ関数を**デストラクタ**（destructor）といいます。第13章で学んだコンストラクタは、オブジェクトの作成時に自動的に呼ばれるのでした。デストラクタはコンストラクタとちょうど反対の機能をもっています。

構文　デストラクタの定義

```
クラス名::~クラス名()
{
    ...
}
```

デストラクタは引数と戻り値をもちません

チルダ(~)をつけます

オブジェクトが破棄されるときの処理を記述します

~Car()

図15-7　デストラクタ

デストラクタは、オブジェクトが破棄される場合に自動的に呼び出されます。デストラクタ内では、確保したメモリの解放などを行うことができます。

デストラクタは引数をもちません。また、戻り値ももちません。関数名（デストラクタの名前）はクラス名の前に~（チルダ）をつけたものを使います。デストラクタを定義し、ここでメモリを解放する処理を書いておくと、オブジェクトの破棄と同時にクラス内で確保したメモリが解放されることになります。

それでは、デストラクタを定義したコードをみてみましょう。

Sample2.cpp ▶ デストラクタを定義する

```
#include <iostream>
#include <string>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
      char* pName;
   public:
      Car::Car(char* pN ,int n, double g);
      ~Car();  ●―― デストラクタの宣言です
      void show();
};

//Carクラスメンバ関数の定義
Car::Car(char* pN ,int n, double g)
{
   pName = new char[strlen(pN)+1];
   strcpy(pName, pN);  ●―― コンストラクタ内で動的に
                            メモリを確保しています
   num = n;
   gas = g;
   cout << pName << "を作成しました。¥n";
}
Car::~Car()  ●―― デストラクタの定義です
{
   cout << pName << "を破棄します。¥n";
   delete[] pName;  ●―― メモリを解放する処理が必要です
}
void Car::show()
{
   cout << "車のナンバーは" << num << "です。¥n";
   cout << "ガソリン量は" << gas << "です。¥n";
   cout << "名前は" << pName << "です。¥n";
```

```
}

//Carクラスの利用
int main()
{
   Car car1("mycar", 1234, 25.5);
   car1.show();

   return 0;
}
```

Sample2の実行画面

```
mycarを作成しました。
車のナンバーは1234です。
ガソリン量は20.5です。
名前はmycarです。
mycarを破棄します。
```


Sample2ではコンストラクタ内で動的にメモリを確保しています。このため、デストラクタにメモリを解放する処理を記述したのです。このように、デストラクタではオブジェクトを破棄する際に必要な処理を記述します。なお、このように、メモリ解放などの終了処理が必要ない場合には、デストラクタを記述する必要はありません。

重要

オブジェクトを破棄する際に必要な処理は、デストラクタに記述する。

デストラクタとコンストラクタ

第13章では異なる引数をもつコンストラクタをオーバーロードできることを学びました。しかし、デストラクタは引数をもたないので、ただ1つの形式しかもちません。したがって、デストラクタを複数定義すること、すなわちデストラクタをオーバーロードすることはできません。

オブジェクトの初期化や代入を正しく行うには？

ところで、実はSample2のCarクラスは、その利用のしかたによっては困ったことがおきる場合があります。たとえば、次のようにクラスを利用する場合を考えてみましょう。

```
Car car1 = mycar;
...
```

オブジェクトを初期化しています

これは、Car型のmycarの値でCar型のcar1を初期化を行うコードです。また、次のような利用法もみてください。

```
Car car2;
car2 = mycar;
```

オブジェクトにほかのオブジェクトを代入しています

このコードでは、mycarの値をcar2に代入しています。

このような代入や初期化を行うと、mycarのメンバがcar1、car2の各メンバにコピーされます。ところが、このコピーによって、mycarのpNameとcar1・car2のpNameは同じメモリをさすようになってしまうのです。これでは各オブジェクトを別のモノとして扱うことができなくなってしまいます。

図 15-8　メンバの単純なコピー
ほかのオブジェクトによって初期化や代入が行われるとき、メンバの単純なコピーが行われると、不具合がおきる場合あります。ここではpNameがコピーされることによって、オブジェクトのメンバが同じメモリをさしてしまっています。

初期化と代入

初期化・代入の際には、どちらもオブジェクトのメンバのコピーが行われ、どちらも＝記号が使われます。しかし、C++では代入と初期化は別の機能として区別されています。これからみるように、初期化の場合にはコピーコンストラクタが呼び出され、代入の場合には、代入演算子による処理が行われます。

コピーコンストラクタを定義する

そこで、ほかのオブジェクトによって、Carクラスのオブジェクトの初期化・代入が行われる場合には、単純にメンバをコピーせず、各メンバ用のメモリを確保する処理が必要となります。

初期化と代入において、正しくメモリの確保と解放が行われるようにクラスを記述してみることにしましょう。

まず、初期化の場合についてみてみましょう。メモリを正しく確保するためには、**コピーコンストラクタ**（copy constructor）という特殊なコンストラクタを定義します。コピーコンストラクタは、オブジェクトがほかのオブジェクトで初期化される場合に呼び出されるコンストラクタです。

構文　コピーコンストラクタの定義（クラス宣言外で行う場合）

```
クラス名::クラス名(const 参照型 引数)
{
   ...
}
```

（const 参照型 引数）：オブジェクトがほかのオブジェクトの値で初期化される際に行わなければならない処理を記述します

実際のコピーコンストラクタは、次のようになります。

```
Car::Car(const Car& c)
{
   cout << c.pName << "で初期化します。¥n";
   pName = new char[strlen(c.pName)+strlen("のコピー1")+1];
   strcpy(pName, c.pName);
   strcat(pName, "のコピー1");
   num = c.num;
   gas = c.gas;
}
```

- Car::Car(const Car& c)：コピーコンストラクタです
- new：コピー先のオブジェクトのためにメモリを確保します
- strcpy(pName, c.pName); / strcat(pName, "のコピー1");：確保した場所に名前を格納します

コピーコンストラクタ内では、コピー先のオブジェクトのために、新しくメモリを確保します。そして、この領域に名前を格納しているのです。

これで、初期化を行ったときにも正しくメモリが確保されるようになります。

```
Car car1 = mycar;
```

初期化を行ったときにコピーコンストラクタが呼び出され・・・

car1 のメンバのためにメモリが確保されます

代入演算子をオーバーロードする

初期化をするときに正しくメモリの確保を行う方法がわかりました。次に、代入をするときに正しくメモリの確保を行うようにしてみましょう。この場合には、代入演算子=をオーバーロードして、正しくメモリを確保します。

構文　代入演算子のオーバーロード

```
クラス名& クラス名::operator=(const 参照型 引数)
{
...
}
```

オブジェクトにほかのオブジェクトの値が代入される際に行わなければならない処理を記述します

代入演算子のオーバーロードはこの章の最初で学んだ、演算子のオーバーロードと同じ方法です。ただし、代入演算子は、必ずメンバ関数としてオーバーロードしなければなりません。

```
Car& Car::operator=(const Car& c)
{
   cout << pName << "に" << c.pName << "を代入します。¥n";
   if(this != &c){
      delete[] pName;
      pName = new char[strlen(c.pName)+
                       strlen("のコピー2")+1];
      strcpy(pName, c.pName);
      strcat(pName, "のコピー2");
      num = c.num;
      gas = c.gas;
   }
  return *this;
}
```

代入演算子のオーバーロードです

コピー先のオブジェクト自身による代入を除きます

コピー先のオブジェクトがすでに確保しているメモリを解放します

コピー先のオブジェクトのためにメモリを確保します

代入演算子のオーバーロードでは、すでに存在するオブジェクトを考慮しなければならないため、コピーコンストラクタの処理よりも複雑です。まずthisポインタを調べて、コピー先のオブジェクト自身による代入である場合を除きます。そして、コピー先のオブジェクトがコンストラクタ内ですでに確保していたメモリを破棄し、新しくメモリを確保するようにしています。

つまり、代入を行ったときに正しくメモリが確保されるのです。

```
Car car2;
car2 = mycar;
```

代入を行ったときに代入演算子関数が呼び出され・・・

car2のメンバのためにメモリが確保されます

それでは、実際にこれらの初期化・代入がうまく行われるようなコードの記述をみてみましょう。

Sample3.cpp ▶ コピーコンストラクタ・代入演算子のオーバーロード

```
#include <iostream>
#include <string>
using namespace std;

//Carクラスの宣言
class Car{
   private:
      int num;
      double gas;
      char* pName;
   public:
      Car(char* pN ,int n, double g);
      ~Car();
      Car(const Car& c);
      Car& operator=(const Car& c);
};

//Carクラスメンバ関数の定義
Car::Car(char* pN ,int n, double g)
{
   pName = new char[strlen(pN)+1];
   strcpy(pName, pN);
```

コピーコンストラクタの宣言です

代入演算子の宣言です

コンストラクタ内で動的にメモリを確保しています

```
    num = n;
    gas = g;
    cout << pName << "を作成しました。¥n";
}
Car::~Car()
{
    cout << pName << "を破棄します。¥n";
    delete[] pName;
}
Car::Car(const Car& c)
{
    cout << c.pName << "で初期化します。¥n";
    pName = new char[strlen(c.pName) + strlen("のコピー1")+1];
    strcpy(pName, c.pName);
    strcat(pName, "のコピー1");
    num = c.num;
    gas = c.gas;
}
Car& Car::operator=(const Car& c)
{
    cout << pName << "に" << c.pName << "を代入します。¥n";
    if(this != &c){
        delete[] pName;
        pName = new char[strlen(c.pName)+strlen("のコピー2")+1];
        strcpy(pName, c.pName);
        strcat(pName, "のコピー2");
        num = c.num;
        gas = c.gas;
    }
    return *this;
}

int main()
{
    Car mycar("mycar", 1234, 25.5);

    Car car1 = mycar;

    Car car2("car2", 0, 0);
    car2 = mycar;

    return 0;
}
```

デストラクタ内でメモリを解放します

コピーコンストラクタです

代入演算子のオーバーロードです

初期化しています

代入しています

Sample3の実行画面

```
mycarを作成しました。
mycarで初期化します。
car2を作成しました。
car2にmycarを代入します。
mycarのコピー2を破棄します。
mycarのコピー1を破棄します。
mycarを破棄します。
```


このコードでは、コピーコンストラクタと代入演算子の定義の中で、新しくメモリを確保し、それぞれのオブジェクトのメンバpNameが異なるメモリをさすようにしました。

初期化・代入が行われる場合

C++のコードの中では、初期化や代入が行われる場合があります。このとき、オブジェクトが動的にメモリを確保するなどの処理を行っている場合は、これまでみてきたように、コピーコンストラクタや代入演算子を定義する必要があります。コピーコンストラクタや代入演算子の中で、メモリを正しく確保しなければならないのです。

なお、C++では次のような場合にも初期化や代入が行われます。たとえば、以下のような場合には、コピーコンストラクタや代入演算子の定義を行うことを検討しなければなりません。

関数に実引数を渡す場合

```
Car mycar;
func1(mycar);
...

//func1関数の定義
void func1(Car c)
{
```

```
    ...
}
```

関数から戻り値が返される場合

```
Car mycar;
mycar = func2();
...

//func2関数の定義
Car func2()
{
    Car c;
    ...
    return c;
}
```

戻り値によって代入が行われます

重要

メンバの単純なコピーが行えない場合には、コピーコンストラクタ・代入演算子関数を定義する。

15.4 テンプレートクラス

テンプレートクラスのしくみを知る

さて、この節では**テンプレートクラス**（template class）について学びます。第7章では、いろいろな型を扱う関数テンプレートを作成できることを学んだことを思い出してください。クラスについても、同じように、

クラスの“ひながた”から、いろいろな型を扱うクラスを作成できる

という機能があります。このひながたのことを**クラステンプレート**（class template）といいます。

クラステンプレートの記述は次のようなものです。

構文 **クラステンプレート**

```
template<テンプレート引数のリスト>
class クラス名{
    ...
};
```

置きかえる型名をTなどとしておきます

クラスの宣言を記述します

構造はクラスの宣言と似ていますね。ただし、クラステンプレートでは、クラス中で扱う型名を特定せずに**テンプレート引数**として記述することができます。

クラステンプレートを記述すると、次のように具体的な型名を与えて、その型を扱うクラスとオブジェクトを作成することができます。

構文 **クラステンプレートからオブジェクトを作成**

```
クラス名<型名> 識別子;
```

クラステンプレートを利用する

実際にクラステンプレートを作成し、利用するコードをみてみましょう。

Sample4.cpp ▶ クラステンプレートを利用する

```
#include <iostream>
using namespace std;

//Arrayクラステンプレート
template <class T>
class Array{
   private:
      T data[5];
   public:
      void setData(int num, T d);
      T getData(int num);
};

template <class T>
void Array<T>::setData(int num, T d)
{
   if(num < 0 || num > 4 )
      cout << "配列の範囲をこえています。¥n";
   else
      data[num] = d;
}
template <class T>
T Array<T>::getData(int num)
{
   if(num < 0 || num > 4 ){
      cout << "配列の範囲をこえています。¥n";
      return data[0];
   }
   else
      return data[num];
}

int main()
{
   cout << "int型の配列を作成します。¥n";
   Array<int> i_array;
   i_array.setData(0, 80);
```

クラステンプレートです

クラステンプレートのメンバ関数の定義です

クラステンプレートのメンバ関数の定義です

int 型を扱うクラス・オブジェクトを作成します

```
    i_array.setData(1, 60);
    i_array.setData(2, 58);
    i_array.setData(3, 77);
    i_array.setData(4, 57);

    for(int i=0; i<5; i++)
       cout << i_array.getData(i) << '¥n';

    cout << "double型の配列を作成します。¥n";
    Array<double> d_array;
    d_array.setData(0, 35.5);
    d_array.setData(1, 45.6);
    d_array.setData(2, 26.8);
    d_array.setData(3, 76.2);
    d_array.setData(4, 85.5);

    for(int j=0; j<5; j++)
       cout << d_array.getData(j) << '¥n';

    return 0;
}
```

double型を扱うクラス・オブジェクトを作成します

Sample4の実行画面

```
int型の配列を作成します。
80
60
58
77
57
double型の配列を作成します。
35.5
45.6
26.8
76.2
85.5
```

メンバ関数の定義が少しややこしいので、注意してみておきましょう。クラステンプレートのメンバ関数の定義は、次のようになっています。

構文 **クラステンプレートのメンバ関数の定義**

```
template <テンプレート引数のリスト>
戻り値の型 クラス名<型名のリスト>::関数名(引数のリスト)
{
   ...
}
```

このクラステンプレートは5つの配列要素を扱うクラスとして定義しています。このクラステンプレートから、int型とdobule型を扱うクラスやオブジェクトを作成し、利用しています。ここでは、Array<型>と指定することで、クラス・オブジェクトを作成しているのです。

このように、クラステンプレートを使うと、さまざまな型を扱うクラス・オブジェクトをかんたんに作成することができるようになります。

なお、C++の標準ライブラリ（第9章、第10章）には、クラステンプレート・関数テンプレートが数多く用意されています。これらは**標準テンプレートライブラリ**（Standard Template Library、STL）と呼ばれています。

重要

クラステンプレートを使うと、扱う型が異なるクラス・オブジェクトをかんたんに作成できる。

15.5 例外処理

例外処理のしくみを知る

最後に、C++の**例外処理**（exception handling）という機能を学ぶことにしましょう。例外処理は、プログラムの実行時におきるさまざまなエラーを処理する機能です。例外処理は、次のような構文になっています。

構文 例外処理

```
try{
    例外を検出する処理
    throw  例外;        ← 例外を送出します
}
catch(型){
    例外時の処理;       ← 送出された例外の型とこの型が一致すれば、ブロック内の処理を行います
}
```

実際に例外処理を記述してみましょう。次のコードをみてください。

Sample5.cpp ▶ 例外処理を行う

```
#include <iostream>
using namespace std;

int main()
{
   int num;
   cout << "1～9までの数を入力してください。¥n";
   cin >> num;

   try{
      if(num <= 0)
```

```
        throw "0以下を入力しました";
    if(num >= 10)
        throw "10以上を入力しました";

    cout << num << "です。¥n";
  }

  catch(char* err){
    cout << "エラー:" << err << '¥n';
    return 1;
  }
  return 0;
}
```

例外（ここでは文字列）を送出します

例外処理を行います

例外処理を行うには、エラーを検出する処理を**tryブロック**でかこんでおきます。エラーなどがおきた場合は、このブロックの中で、

ある値を「例外」（exception）として送出する

という処理を行います。例外を送出するには、**throw**というキーワードを使います。tryブロックのあとには、**catchブロック**を記述します。例外が送出されると、catchブロックで例外を受けとって処理をするようになっています。

このようなコードを記述すると、エラー処理をcatchブロックにまとめて記述することができます。

Sample5を実行し、キーボードから負の数を入力した場合は、次のように出力されます。

Sample5の実行画面

```
1～9までの数を入力してください。
-1 ↵
エラー：0以下を入力しました。
```

誤った入力を行いました

例外が送出され、例外処理が行われました

たしかに、catchブロック内の処理が行われていますね。

Lesson 15

重要

throw文で例外を送出する。

```
int main()
{
   try{
      if(num <= 0)
         throw "0以下を入力しました";
      cout << num << "です。¥n";
   }

   ...

   catch(char* err){
      cout << "エラー:" << err << '¥n';
      return 1;
   }
   return 0;
}
```

図15-9 例外処理
tryブロックの中で送出された例外を、catchブロックで受けとって処理することができます。

例外処理の高度な機能

例外処理にはこのほかにもさまざまな高度な機能があります。まず、さまざまな型の値を例外として複数送出することができます。送出した値によって、きめ細かい処理を行うことができるのです。この場合には、複数の型を複数のcatchブロックで受けとるようにします。

```
try{
   throw 例外1;
   ...
   throw 例外2;
}

catch(例外1の型){
   例外1が送出されたときの処理
```

```
}
catch(例外2の型){        ● 例外2の場合はこちらの処理が行われます
    例外2が送出されたときの処理
}
```

また、例外は、tryブロック内から呼び出された関数からも送出することができます。大規模なプログラム場合は、関数やクラスを設計する人が、例外を送出することだけを記述しておきます。そして、それらの関数やクラスを利用する人が、例外が送出された場合のエラー処理を記述するのです。このしくみを利用すると、エラーを柔軟に処理できるようになっています。

```
//func関数の定義
func(){
    ...
    throw 例外;        ● 関数を定義する際は例外を送出するだけにします
}

//func関数の利用
...
try{
    func();
}
catch(型){        ● 関数を利用する人がエラー時の処理を記述します
    例外処理
}
```

重要

プログラムの実行時におこるエラー処理を、例外処理として記述できる。

15.6 レッスンのまとめ

この章では、次のようなことを学びました。

- 演算子をオーバーロードすると、オブジェクトに演算子を使うことができます。
- 基本型への型変換を行うには、変換関数・変換コンストラクタを定義します。
- デストラクタは、オブジェクトを破棄する際に呼び出されます。
- コピーコンストラクタは、オブジェクトがほかのオブジェクトの値で初期化されるときに呼び出されます。
- メンバの単純なコピーが行われては困る場合には、デストラクタ・コピーコンストラクタ・代入演算子をオーバーロードします。
- クラステンプレートから、さまざまな型を扱うクラスを作成することができます。
- 送出される例外の型にしたがって、例外処理を行うことができます。

この章では、クラスに関するさまざまなトピックを学びました。高度な機能も含まれていますが、1つずつしっかりとおさえていきましょう。

練習

1. 次の項目について○か×で答えてください。

① 演算子は、メンバ関数としてもフレンド関数としても定義できる場合がある。
② すべての演算子をオーバーロードできるわけではない。
③ 単項演算子をオーバーロードして２項演算子とすることができる。

2. 次の項目について○か×で答えてください。

① クラス宣言内には、複数のデストラクタ宣言を記述することができる。
② デストラクタは戻り値と引数をもたない。
③ オブジェクトにほかのオブジェクトの値を代入する際には、コピーコンストラクタが呼び出される。

3. 15.1節のPointクラスのコードに次の演算子をオーバーロードするコードを記述してください。

-演算子
--演算子

4. クラスを使って、上記２つの演算子を利用するコードを記述してください。

Lesson 16

ファイルの入出力

これまでのプログラムの中には、処理結果を画面に出力したり、キーボードからの入力を受けとるものがありました。本書の最後となるこの章では、画面・キーボードなどの入出力に関する機能について、くわしく学ぶことにしましょう。C++では、この知識をファイルを扱う場合にも応用することができます。この章を学ぶことで、より実用的なプログラムを作成していくことにしましょう。

Check Point!

- ストリーム
- 挿入演算子
- 抽出演算子
- マニピュレータ
- 書式状態フラグ
- ファイル入出力
- コマンドライン引数

16.1 ストリーム

ストリームのしくみを知る

いままで作成してきたプログラムの中には、画面に文字や数値を出力したり、キーボードから情報を入力する処理をするものがありました。入出力は、画面やキーボード、それにファイルに対して行われます。これらの装置は一見異なるものですが、C++では、さまざまな装置に対する入出力を統一的な方法で扱うことができるようになっています。この入出力機能をささえる概念を、**ストリーム**（stream）といいます。

図16-1　ストリーム
入出力はストリームの概念を使って行われます。

ストリームは、さまざまな異なる装置を、同じように扱うための抽象的なしくみです。この章で、さまざまな入出力を行うプログラムを作成していくことにしましょう。

C++ストリームの機能は、**istreamクラス**、**ostreamクラス**として、標準ライブラリ<iostream>などで提供されています。

いままで入出力の際に使ってきたcinとcoutは、次のクラスのオブジェクトです。

- cin ……istreamクラスのオブジェクト
- cout ……ostreamクラスのオブジェクト

私たちはこれらのクラスや作成したオブジェクトを利用して、ストリームによる入出力機能を利用することができるようになっているのです。

入出力ストリームを利用する

それではまず、cinとcoutを使ったコードを復習してみましょう。

これまでのコードで作成してきたように、cinと>>記号を使えば、キーボードからの入力を変数などに読み込むことができました。これは、istreamクラスで>>演算子がこのような機能をもつものとしてオーバーロードされているからです。

演算子をオーバーロードできることは、第15章で学んでいます。istreamクラスでは、>>演算子のオーバーロードが行われているのです。この入力用にオーバーロードされた**>>**演算子を、特に**抽出演算子**（extraction operator）と呼びます。

また、<<演算子については、ostreamクラスで標準出力に文字や数値を送ることができるようにオーバーロードされています。このような**<<**演算子を**挿入演算子**（insertion operator）と呼びます。

Sample1.cpp ▶ >>演算子を使う

```
#include <iostream>
using namespace std;

int main()
{
   int i;
   double d;
   char str[100];

   cout << "整数値を入力してください。¥n";
   cin >> i;
   cout << "小数値を入力してください。¥n";
   cin >> d;
   cout << "文字列を入力してください。¥n";
   cin >> str;

   cout << "入力した整数値は" << i << "です。¥n";
   cout << "入力した小数値は" << d << "です。¥n";
   cout << "入力した文字列は" << str << "です。¥n";

   return 0;
}
```

いろいろな型の入出力が行えます

Lesson 16

Sample1 の実行画面

```
整数値を入力してください。
5 ↵
小数値を入力してください。
2.6 ↵
文字列を入力してください。
Hello ↵
入力した整数値は5です。
入力した小数値は2.6です。
入力した文字列はHelloです。
```

<<演算子、>>演算子はデータの型を意識せずに使うことができるようにオーバーロードされています。

挿入演算子をオーバーロードする

それでは、私たちが定義した新しい型であるクラスを、<<演算子・>>演算子を使って入出力することはできるのでしょうか。このときには第15章で学んだ方法で、<<演算子や>>演算子の新しい用法をオーバーロードすることになります。たとえば、第14章のCarクラスオブジェクトを出力する<<演算子を、次のようにフレンド関数としてオーバーロードします。

```
#include <iostream>
...
class Car {
   ...
   friend ostream& operator<<(ostream& out, Car& c);
   };
   ...
ostream& operator<<(ostream& out, Car& c)
{
   out << "ナンバー" << num << ":" << "ガソリン" << gas;
   return out;
}
...
```

このように<<演算子をオーバーロードすることで、Carクラスのオブジェクトをそのまま出力できることになります。

```
//Carクラスの利用
int main()
{
   Car mycar(1234, 25.5);
   cout << mycar;
}
```

Carクラスのオブジェクトに<< 演算子を使えます

上のコードの出力結果は次のようになります。

```
ナンバー1234:ガソリン25.5
```

<<演算子をオーバーロードしなければ、Carクラスのオブジェクトに対して<<演算子を使うことはできないことに注意してください。

なお、<<演算子は左結合であるため、ostreamクラスへの参照を戻り値としています。また、左オペランドがostreamクラスのオブジェクトとなりますから、Carクラスのメンバ関数とすることはできません。

1文字の入出力を行う

ところで、Sample1の実行中に、空白を含んだ文字列を入力してみてください。すると、空白以降の文字が読み込まれていないことがわかります。

Sample1の実行画面その2

```
整数値を入力してください。
5 ↵
小数値を入力してください。
2.6 ↵
文字列を入力してください。
This is a pen. ↵
```

空白を入力してみました

```
入力した整数値は5です。
入力した小数値は2.6です。
入力した文字列はThisです。
```
空白以降が読み込まれていません

これは、<<演算子が空白を読み込まないようになっているためです。そこで、ここでは、空白を含む1文字を読み込む機能をもつistreamクラスの**get()**メンバ関数をおぼえましょう。また、ostreamクラスには1文字を出力する**put()**メンバ関数があります。2つをあわせて呼び出してみることにします。

Sample2.cpp ▶ 1文字入出力関数を使う

```
#include <iostream>
using namespace std;

int main()
{
   char ch;

   cout << "文字を続けて入力してください。¥n";

   while(cin.get(ch)){
      cout.put(ch);
   }

   return 0;
}
```
1文字ずつ読み込みます

1文字ずつ出力します

Sample2の実行画面

```
文字を続けて入力してください。
This is a pen. ↵
This is a pen.
Ctrl+Z
```
空白も読み込まれます

で終了します

今度は空白が読み込まれています。このプログラムは、Windows（MS-DOS）では、Ctrl+Z で終了します。UNIXなどでは、Ctrl+D で終了します。

構文　**1文字の入出力**

```
get(変数);
put('文字');
```

1文字入力します
1文字出力します

出力幅を指定する

そのほかにも、istreamクラスとostreamクラスのメンバ関数には入出力に関するさまざまな機能が定義されています。いくつかとりあげてみることにしましょう。

まず、ostreamクラスの**width()メンバ関数**を使ってみましょう。width()関数を使うと、出力幅を指定することができます。

構文　**出力幅の指定**

```
width(出力幅);
```

Sample3.cpp ▶ width()関数を使う

```
#include <iostream>
using namespace std;

int main()
{
   for (int i=0; i<=10; i++){
      cout.width(3);
      cout << i;
   }
   cout << '¥n';

   return 0;
}
```

出力幅を3とします

Sample3の実行画面

```
  1  2  3  4  5  6  7  8  9 10
```

3文字の幅で出力されています

Lesson 16

1から10までの数値を繰り返して出力しています。3文字幅で出力が行われていますね。

```
  1  2  3  4  5
```

図16-2 出力幅の指定
width()関数を使うと出力幅の指定ができます。

指定した幅よりも出力文字列の幅が大きい場合

指定した幅よりも出力する文字列や数値の幅が大きいときは、出力する文字列や数値にあわせた幅で出力されます。

フィル文字を指定する

width()関数を使って出力幅を指定すると、幅に満たない場合に空白が挿入されます。このとき、**fill()関数**を使うと、空白を指定文字で埋めることができます。

構文 **フィル文字の指定**

```
fill('文字')
```

次のようなコードを入力してみましょう。

Sample4.cpp ▶ fill()関数を使う

```
#include <iostream>
using namespace std;

int main()
{
    for (int i=0; i<=10; i++){
```

```
        cout.width(3);      // 出力幅を3とします
        cout.fill('-');     // フィル文字を-とします
        cout << i;
    }
    cout << '¥n';

    return 0;
}
```

Sample4の実行画面

```
--1--2--3--4--5--6--7--8--9--10
```

–が出力幅にあわせて埋められます

このコードは、Sample3をfill()関数を使って改変したものです。出力幅にあわせて文字'–'が埋め込まれています。'–'には適当な文字を指定することができますので、ためしてみてください。

--1--2--3--4--5

図16-3 フィル文字の指定
fill()関数を使うと、フィル文字の指定ができます。

数値の精度を指定する

浮動小数点数を出力するときは、有効桁数（精度）を指定する**precision()関数**を使うことができます。

構文 **精度の指定**

```
precision(精度指定)
```

Lesson 16

Sample5.cpp ▶ precision()関数を使う

```
#include <iostream>
using namespace std;

int main()
{
    double pi = 3.141592;
    int num;

    cout << "円周率を出力します。¥n";
    cout << "有効桁何桁で出力しますか？(1～7)¥n";
    cin >> num;

    cout.precision(num);  // 出力の際の有効桁を指定します

    cout << "円周率は"<< pi <<"です。¥n";

    return 0;
}
```

Sample5の実行画面

```
円周率を出力します。
有効桁何桁で出力しますか？（1～7）
3 ↵
円周率は3.14です。  // 指定した有効桁で出力されます
```

このコード中では、円周率をあらわすpiに3.141592を代入しています。そのあとで、ユーザーに有効桁数を入力させて、その精度で円周率を出力しています。

3.141592

図16-4 精度指定
precision()関数を使うと、精度の指定ができます。

書式設定の範囲

ここでは、ostreamクラスのメンバ関数による書式設定を紹介しました。この書式設定は、最初に行う出力だけに有効です。二度目以降の出力で書式を指定したい場合は、再度メンバ関数を呼び出す必要があります。

次節では、マニピュレータによる書式設定方法を紹介します。マニピュレータによる方法では何度も書式設定をする必要はありません。

書式状態フラグを設定する

このほか、**書式状態フラグ**というものを使ってさまざまな書式設定を行うことができます。例として、出力位置を揃えるコードを記述してみましょう。

Sample6.cpp ▶ 書式状態フラグの設定

```
#include <iostream>
using namespace std;

int main()
{
   cout.setf(ios::left,ios::adjustfield);    // 左寄せで出力します
   for (int i=0; i<=5; i++){
      cout.width(5);
      cout.fill('-');
      cout << i;
   }
   cout << '¥n';
   cout.unsetf(ios::left);    // 左寄せを解除します
   cout.setf(ios::right,ios::adjustfield);
   for (int j=0; j<=5; j++){    // 右寄せで出力します
      cout.width(5);
      cout.fill('-');
      cout << j;
   }
   cout << '¥n';
```

```
    return 0;
}
```

Sample6 の実行画面

```
0----1----2----3----4----5----
----0----1----2----3----4----5
```

左寄せで出力されます

右寄せで出力されます

書式状態フラグは**setf()メンバ関数**を呼び出すことで設定できます。「ios:: ・・・」という指定が書式状態フラグです。このようにios::leftを設定すると、出力幅内で左寄せで出力されます。

また、**unsetf()メンバ関数**の呼び出しをすると、書式の設定が解除されます。このコードではさらに、ios::rightを設定することで、右寄せで出力しています。

構文 **書式状態フラグの設定**

```
setf(書式状態フラグ)
```

構文 **書式状態フラグの解除**

```
unsetf(書式状態フラグ)
```

書式状態フラグは、次の表のようになっています。

表16-1 ：書式状態フラグ

書式状態フラグ	内容
ios::adjustfield	出力位置を設定する
ios::basefield	基数を設定する
ios::floatfield	小数の記法を設定する
ios::skipws	空白文字をスキップする
ios::left	指定した幅内で左寄せにする
ios::right	指定した幅内で右寄せにする
ios::internal	符号を左寄せ、数値を右寄せにする

書式状態フラグ	内容
ios::dec	10 進数にする
ios::oct	8 進数にする
ios::hex	16 進数にする
ios::showbase	8 進数の前に 0、16 進数の前に 0x をつける
ios::showpoint	末尾に 0 をつける
ios::showpos	符号をつける (+ を表示する)
ios::scientific	科学表記法とする (e を使う)
ios::fixed	固定小数点形式とする
ios::uppercase	英字を大文字にする

16.2 マニピュレータ

ここまで、入出力ストリームクラスのメンバ関数を使って書式の設定を行う方法をみてきました。標準ライブラリでは、入出力のための書式を設定する特別な関数が定義されています。この特別な関数を、**マニピュレータ**（manipulator）といいます。この節では、よく使うマニピュレータを学んでいきます。

改行を出力する

改行を行うマニピュレータがendlです。このマニピュレータは、エスケープシーケンス'¥n'の出力のかわりに使うことができます。

構文 **改行の出力**

```
endl
```

それでは、endlを'¥n'のかわりに使ってみましょう。

Sample7.cpp ▶ endlを使用する

```
#include <iostream>
using namespace std;

int main()
{
    cout << "こんにちは！" << endl;
    cout << "さようなら！" << endl;

    return 0;
}
```

ここで改行されます

Sample7 の実行画面

```
こんにちは！
さようなら！
```

改行されています

endl を記述すると、'¥n' 文字の出力と同じように改行されているのがわかりますね。

10 進数以外で数値を出力する

マニピュレータを使って、10 進数以外の表記で数値を出力することができます。次の表をみてください。

表 16-2 ：基数を変更するマニピュレータ

dec	書式を 10 進数に設定します
oct	書式を 8 進数に設定します
hex	書式を 16 進数に設定します

次のコードは、マニピュレータを使って、10 進数以外の表記で数値を出力しています。

Sample8.cpp ▶ いろいろな表記法で出力する

```
#include <iostream>
using namespace std;

int main()
{
   cout << "10を10進数で表記すると" << dec << 10 << "です。¥n";
   cout << "10を8進数で表記すると" << oct << 10 << "です。¥n";
   cout << "12を8進数で表記すると" << 12 << "です。¥n";
   cout << "10を16進数で表記すると" << hex << 10 << "です。¥n";

   return 0;
}
```

いろいろな表記法で出力します

指定しなかった場合は、最後に指定した書式が有効となります

Lesson 16

Sample8の実行画面

```
10を10進数で表記すると10です。
10を8進数で表記すると12です。
12を8進数で表記すると14です。
10を16進数で表記するとaです。
```

10進数以外の表記でも出力できます

マニピュレータdecは10進数として出力します。また、octは8進数、hexは16進数による出力を行います。なお、マニピュレータは一度指定すると、指定を変更するまでその指定が使われます。そのため、3行目の出力は8進数で出力されています。

出力幅を指定する

マニピュレータを使って、16.1節のメンバ関数と同様の書式設定を行うこともできます。ここでは、出力幅の指定を行う**setw()**マニピュレータを使ってみましょう。ただし、このマニピュレータを使用するには、<iomanip>をインクルードする必要があります。

Sample9.cpp ▶ 出力幅の指定

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   for (int i=0; i<=10; i++){
      cout << setw(3) << i;
   }
   cout << '¥n';

   return 0;
}
```

<iomanip>をインクルードします

出力幅を3とします

Sample9 の実行画面

3文字の幅で出力されています

```
  1  2  3  4  5  6  7  8  9 10
```

setw()マニピュレータを使うと、width()メンバ関数を使ったときと同様の書式設定を行うことができます。

メンバ関数とマニピュレータ

このほか、入出力クラスのメンバ関数を呼び出して行った設定を、次のマニピュレータで行うことができます。マニピュレータでは、設定した書式は変更しない限り保持されるので便利です。ただし、<iomanip> をインクルードすることを忘れないでください。

フィル文字の指定 …… setfill(文字)
精度の指定 …… setprecision(整数値)

16.3 ファイル入出力の基本

ファイル入出力のしくみを知る

これまでみてきたように、C++では、ストリームの概念を使って、画面やキーボードへの入出力をすることができました。C++では、ファイルへの入出力（データの読み書き）もストリームへの入出力として扱うことができます。ファイルへの書き込みは「出力」、ファイルからの読み込みは「入力」の作業にあたります。この節では、ファイルによる入出力操作の基本を学んでいきましょう。

まず、ファイルを扱う場合の基本手順をおぼえてください。コード上でファイルの処理を行うには、次のような順序で操作を行います。

ファイルをオープンする
↓
ファイルを読み書きする
↓
ファイルをクローズする

ファイルの「オープン」や「クローズ」とは、操作をはじめる前に入出力に使われるストリームの概念と実際のファイルを結びつけ、操作が終わったあとに切り離す処理のことをいいます。ファイルを使う際には、最初にオープン、最後にクローズの作業が必要となります。

①オープン　②読み書き　③クローズ

図16-5 ファイル操作の基本
ファイル操作は①オープン②読み書き③クローズの順番で行います。

C++の標準ライブラリでは、ファイルを読み書きするために次のようなクラス提供しています。これらのクラスはファイル入出力に必要な機能を定義しているのです。

- ostreamクラスから派生したofstreamクラス …… ファイルへの書き出し
- istreamクラスから派生したifstreamクラス …… ファイルからの読み込み

ofstreamクラスやifstreamクラスを利用するには、標準ライブラリ<fstream>をインクルードする必要があります。

では、ファイルを扱うコードを作成してみることにしましょう。

Sample10.cpp ▶ ファイルの基本操作をする

```
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
   ofstream fout("test0.txt");
   if(!fout){
      cout << "ファイルをオープンできませんでした。¥n";
      return 1;
   }
   else
      cout  << "ファイルをオープンしました。¥n";

   fout.close();
   cout << "ファイルをクローズしました。¥n";
```

<fstream>をインクルードします
ファイルをオープンします
オープンできないときはエラー処理をします
ファイルをクローズします

```
    return 0;
}
```

Sample10の実行画面

```
ファイルをオープンしました。
ファイルをクローズしました。
```

このコードでは、ifstreamクラスからfoutというオブジェクトを作成しています。ifstreamクラスのオブジェクトが作成されると、ファイルがオープンされます。もし、ファイルのオープンができずにエラーとなった場合はfoutがfalseと評価されるため、if文を使ってエラー時の処理を記述しておきます。

このコードでは最後に、ファイルをクローズするためのclose()関数を呼び出してファイルを閉じています。

ストリームの状態を知る

上ではストリームの状態を知るために!foutという条件を記述しました。入出力ストリームの状態を調べるには、このほかにfstreamクラスの基本クラスであるiosクラスのメンバ関数を使う方法もあります。

メンバ関数名	機能
eof()	終わりに到達したかどうかを調べる
fail()	エラーが発生したかどうかを調べる
bad()	エラーが発生したかどうかを調べる
good()	ストリームが正常であるかどうかを調べる
rdstate()	エラーの値を調べる

たとえば、

```
!fout
```

という条件は

```
fout.fail()
```

と記述しても同じです。

本書では、Sample15で、eof()関数を呼び出してファイルの終わりに達したかどうかを調べています。

ファイルに出力する

それではSample10にコードを追加して、実際に、データをファイルに書き込んでみましょう。

Sample11.cpp ▶ ファイルに出力する

```
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
   ofstream fout("test1.txt");
   if(!fout){
      cout << "ファイルをオープンできませんでした。¥n";
      return 1;
   }
   else
      cout << "ファイルをオープンしました。¥n";

   fout << "Hello!¥n";
   fout << "Goodbye!¥n";
   cout << "ファイルに書き込みました。¥n";

   fout.close();
   cout << "ファイルをクローズしました。¥n";
   return 0;
}
```

ファイルにデータを書き込んで（出力して）います

Lesson 16

Sample11の実行画面

```
ファイルを作成しました。
ファイルに書き込みました。
ファイルをクローズしました。
```

このコードでは、ファイルにデータを書き込む処理を追加しています。これをファイルへの出力ともいいます。

オープンしたファイルに結びつけられたfoutに、<<演算子を使って文字列を送り込んでいるのです。coutに対する出力する場合と、ほとんど同じ方法であることがわかるでしょう。

プログラムを実行したらtest1.txtを開いてみましょう。次のようにファイルにデータが書き込まれているでしょうか？

test1.txt

```
Hello!
GoodBye!
```

書式を設定してファイルに出力する

今度は、もう少し扱うデータを増やしてみましょう。キーボードから入力した学生のテストの点数をファイルに書き込んでみることにします。

Sample12.cpp ▶ ファイルに出力する

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
```

```
    ofstream fout("test2.txt");
    if(!fout){
       cout << "ファイルをオープンできませんでした。¥n";
       return 1;
    }

    const int num = 5;
    int test[num];
    cout << num <<"人の点数を入力してください。¥n";
    for(int i=0; i<num; i++){
       cin >> test[i];
    }

    for(int j=0; j<num; j++){
       fout << "No." << j+1 << setw(5) << test[j] << '¥n';
    }

    fout.close();

    return 0;
}
```

データをキーボードから入力します

ファイルにデータを書き込んでいます

画面への出力と同様に書式設定を行うことができます

ここでは、出力する際にsetw()マニピュレータを利用しています。16.1節と16.2節で学んだ各種書式の設定は、ファイルの入出力にも使うことができるのです。画面への出力幅を設定する場合と同じです。コードを実行すると、次のようにtest2.txtというファイルが作成され、にテストの成績が一定の出力幅で書き込まれます。

test2.txt

```
No.1   80
No.2   60
No.3   22
No.4   55
No.5   30
```

ファイルから入力する

では今度は、ファイルからデータを読み込むコードを記述してみましょう。まず、Sample11で作成したファイル（test1.txt）を用意してください。この内容を画面に出力してみることにしましょう。

Sample13.cpp ▶ ファイルから入力する

```
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
   ifstream fin("test1.txt");
   if(!fin){
      cout << "ファイルをオープンできませんでした。¥n";
      return 1;
   }

   char str1[16];
   char str2[16];
   fin >> str1 >> str2;
   cout << "ファイルに書き込まれている2つの文字列は¥n";
   cout << str1 << "です。¥n";
   cout << str2 << "です。¥n";

   fin.close();

   return 0;
}
```

ファイルをオープンします（ifstreamクラスのオブジェクトを作成します）

ファイルからデータを読み込んで（入力して）います

Sample13の実行画面

```
ファイルに書き込まれている2つの文字列は
Hello!です。
Goodbye!です。
```

ファイルからデータを読み込む場合には、入力用ストリームクラスのオブジェ

クトを作成することで、ファイルをオープンします。、Sample11ではistreamクラスからfinというオブジェクトを作成することで、ファイルをオープンしているのです。

オープンしたファイルからデータを読み出すには、キーボードからデータを読み込む方法と同じように、>>演算子を使うことができます。このコードでは、読み込んだデータをcoutを使って画面に出力しました。

ifstreamクラスのオブジェクト >> 変数

図16-6 ファイルからの入力
ファイルからの入力は>>演算子を使って行います。

大量のデータを入力する

ファイルを扱うプログラムはたいへん便利です。たとえば、第9章のテストの成績を処理するコードでは、キーボードから何人もの学生の点数を一人ずつ入力しました。しかし、テストのデータをあらかじめファイルとして用意しておけば、大量のデータを読み込んで、柔軟なコードを記述することができるようになります。

では、ためしてみましょう。まず、次のようなファイルをテキストエディタで作成してみてください。

test3.txt

```
80
68
22
33
56
78
33
56
```

このようなデータを用意しておきます

Lesson 16

これは8人の学生のテストの点数をあらわすデータです。このたくさんのデータを読み込んで、成績処理を行うコードを記述してみましょう。

Sample14.cpp ▶ ファイルから入力する

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   ifstream fin("test3.txt");
   if(!fin){
      cout << "ファイルをオープンできませんでした。¥n";
      return 1;
   }

   const int num = 8;
   int test[num];
   for(int i=0; i<num; i++){
      fin >> test[i];
   }
   int max = test[0];
   int min = test[0];
   for(int j=0; j<num; j++){
      if(max < test[j])
         max = test[j];
      if(min > test[j])
         min = test[j];
      cout << "No." << j+1 << setw(5) << test[j] << '¥n';
   }

   cout << "最高点は" << max <<"です。¥n";
   cout << "最低点は" << min <<"です。¥n";

   fin.close();

   return 0;
}
```

ファイルからデータを読み込んでいます

最高点・最低点を調べます

Sample14の実行画面

```
No.1 80
No.2 68
No.3 22
No.4 33
No.5 56
No.6 78
No.7 33
No.8 56
最高点は80です。
最低点は22です。
```

このコードでは、保存しておいたファイルからデータを読み込み、最高点と最低点などを出力する成績管理を行っています。

ここでは8人のデータを用意していますが、このようにファイルを使えば、大量のデータをあらかじめ用意しておくことができるので、入力作業がたいへん楽になります。

バイナリファイルとテキストファイル

ファイルにはテキストファイルとバイナリファイルがあります。この章で扱ったファイルは、テキストファイルです。テキストファイルには、テキストエディタで簡単に閲覧できるという利点があります。この章の前半でみたように書式の設定を行うこともできます。

これに対してバイナリファイルは、コンピュータの内部で扱われるデータ形式のままが保存されているファイルです。バイナリファイルは、ファイルサイズがコンパクトになる場合があります。たとえば、「1234567」などというint型のデータであれば、ファイルはint型のサイズ（たとえば4バイト）になります。もしこのデータをテキストファイルとして保存すると、7文字のデータ（7バイト）となりますから、ファイルサイズがコンパクトになる場合があるのです。また、データ形式を文字に変換する手間がいらないので、読み書きにかかる時間も速くなります。

また、入出力クラスには、ファイルの先頭部分以外からの読み書きをする機能が提供されています。このようなファイルへのアクセス方法のことを**ランダムアクセス**といいます。ランダムアクセスによって、指定したデータの部分だけを読み書きすることができます。

16.4 コマンドラインからの入力

コマンドライン引数を使う

ところで、これまでのコードでは、読み書きするファイル名はすべて、「test●.txt」などというあらかじめ決められたファイル名を使ってきました。しかしプログラムを実行するときに、読み書きするファイル名をユーザーが指定できれば、より便利なプログラムになることでしょう。

C++では、ユーザーが指定した文字列を実行時に受けとるしくみとして、**コマンドライン引数**（command line argument）という機能を使うことができます。コマンドライン引数は、次のようなmain()関数の引数のかたちで文字列を受けとるものです。

構文 **コマンドライン引数**

```
int main(int argc, char* argv[])
{
  ...
}
```


このコードを実行すると、1番目の引数であるargcには、ユーザーが入力した文字列の個数が格納されます。また、2番目の引数である配列argv[]には、ユーザーが入力した文字列へのポインタが格納されます。

たとえば、ユーザーが次のように入力してプログラムを実行した場合には、argcとargv[]に次のような値が格納されるのです。

```
Sample1 myfile.txt ↵
```

argc	argv[0]	argv[1]
↑	↑	↑
引数の数 （ここでは2）	入力した1番目の文字列の位置 （ここでは「Sample1」）	入力した2番目の文字列の位置 （ここでは「myfile.txt」）

プログラムを実行するとき、1番目に入力するのはプログラム名です。そのあとにスペースで区切って2番目、3番目・・・と文字列を続けて入力すると、それらがmain()関数の引数として渡されるのです。これらの文字列は、char* argv[]という文字列ポインタの配列として扱われます。

さっそくコードを記述してみましょう。今度は次のように、テキストが入力されているmyfile.txtを用意してください。

myfile.txt

```
A long time ago,
There was a little girl.
```

Sample15.cpp ▶ コマンドライン引数を使う

```
#include <fstream>
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
   if(argc != 2){
      cout << "パラメータの数が違います。¥n";
      return 1;
   }

   ifstream fin(argv[1]);
   if(!fin){
      cout << "ファイルをオープンできませんでした。¥n";
      return 1;
   }

   char ch;
   while(!fin.eof()){
      fin.get(ch);
```

- if(argc != 2){ … 入力した文字列の個数を調べます
- ifstream fin(argv[1]); … 入力した2番目の文字列（ファイル名）を指定して、ファイルをオープンします
- while(!fin.eof()){ … ファイルの終わりまで繰り返します
- fin.get(ch); … ファイルから1文字読み込みます

```
        cout.put(ch);   ← 画面に1文字出力します
    }

    fin.close();

    return 0;
}
```

Sample15の実行方法

```
Sample myfile.txt ↵
```

Sample15の実行画面

```
A long time ago,
There was a little girl.
```

ここではプログラムを実行するときに、読み書きするファイル名をプログラム名に続けて入力します。

このプログラムでは、まず引数argcを調べて、正しい数の引数（文字列）が入力されたかチェックしています。ここでは2つの引数が入力されているべきですから、次のようなコードにします。

```
if(argc != 2){          ← argcの値を調べます
    cout << "パラメータの数が違います。¥n";
    return 1;           ← 違っていればプログラムを終了します
}
```

そのあとで、引数のargv[1]（ファイル名をあらわす文字列へのポインタ）を使って、いままでと同じようにファイルをオープンしています。

```
ifstream fin(argv[1]);   ← argv[1]は指定したファイル名です
```

このため、読み込むファイル名が「myfile.txt」ではなくても、プログラムをもう一度作成しなおさずに、別のファイル名を指定することができます。

このようにコマンドライン引数を使えば、ユーザーが実行時に情報を与えるプログラムを作成することができるのです。こうして指定したファイルの内容を1文字ずつ読み出すプログラムを作成することができます。

なお、今回はこれまでと異なり、ファイルの中に何が入っているのかわからないため、eof()関数を使ってファイルの終わりまで読み出すようにしています。

図 16-7 **コマンドライン引数**
main()関数にコマンドライン引数を指定することができます。

コマンドラインから引数を渡すことができる。

16.5 レッスンのまとめ

この章では、次のようなことを学びました。

- 入出力機能を使うには、標準ライブラリ <iostream> を利用します。
- ostream クラスのメンバ関数を使うと、書式設定を行うことができます。
- マニピュレータを使うと、書式設定を行うことができます。
- ファイル入出力を行うには、標準ライブラリ <fstream> を利用します。
- コマンドライン引数を利用すると、プログラムに文字列を渡すことができます。

この章では、入出力に関する機能を学びました。画面やキーボードへの入出力や、ファイルの読み書きを同じように扱うことができたでしょうか。ファイルなどを利用すれば、バリエーションにとんだ実用的なプログラムを作成できるようになるでしょう。さまざまな入出力を行ってみてください。

練習

1. 次のように画面に出力するコードを記述してください。

```
--1--2--3--4--5
--6--7--8--9-10
-11-12-13-14-15
-16-17-18-19-20
-21-22-23-24-25
-26-27-28-29-30
```

2. Sample14のコードを、ファイル名を指定して実行できるようにしてください。

```
Sample14 test3.txt ↵
```

Appendix A

練習の解答

Lesson 1 はじめの一歩

1. ① × ② ○ ③ × ④ × ⑤ ×

Lesson 2 C++ の基本

1. このコードは文法上の間違いはありません。コンパイルし、実行することができます。しかしたいへん読みにくいコードです。改行やインデントなどを行って、次のように読みやすいコードにします。

```
#include <iostream>
using namespace std;

int main()
{
   cout << "こんにちは¥n";
   cout << "さようなら¥n";

   return 0;
}
```

2.

```
#include <iostream>
using namespace std;

int main()
{
   //123と45をわけて表示する
   cout << 1 << 2 << 3 << '¥n' << 4 << 5 << '¥n';

   return 0;
}
```

3.

```
#include <iostream>
using namespace std;

int main()
{
   cout << 123 << '¥n';
   cout << "¥¥100円もらった¥n";
   cout << "またあした¥n";
```

```
    return 0;
}
```

4.

●8 進数

```
#include <iostream>
using namespace std;

int main()
{
    cout << 06 << '¥n';
    cout << 024 << '¥n';
    cout << 015 << '¥n';

    return 0;
}
```

●16 進数

```
#include <iostream>
using namespace std;

int main()
{
    cout << 0x6 << '¥n';
    cout << 0x14 << '¥n';
    cout << 0xD << '¥n';

    return 0;
}
```

Lesson 3 変数

1.

```
#include <iostream>
using namespace std;

int main()
{
    double pi;
```

```
   cout << "円周率の値はいくつですか？¥n";
   cin >> pi;
   cout << "円周率の値は" << pi << "です。¥n";

   return 0;
}
```

2.

```
#include <iostream>
using namespace std;

int main()
{
   char ch;

   cout << "アルファベットの最初の文字は何ですか？¥n";
   cin >> ch;
   cout << "アルファベットの最初の文字は" << ch << "です。¥n";

   return 0;
}
```

3.

```
#include <iostream>
using namespace std;

int main()
{
   double height, weight;

   cout << "身長と体重を入力してください。¥n";
   cin >> height >> weight;
   cout << "身長は" << height << "センチです。¥n";
   cout << "体重は" << weight << "キロです。¥n";

   return 0;
}
```

Lesson 4 式と演算子

1.

```
int main()
{
    int ans1 = 0-4;
    double ans2 = 3.14*2;
    double ans3 = (double)5/(double)3;
    double ans4 = 30%7;
    double ans5 = (7+32)/(double)5;

    cout<< "0-4は" << ans1 << "です。¥n";
    cout<< "3.14×2は" << ans2 << "です。¥n";
    cout<< "5÷3は" << ans3 << "です。¥n";
    cout<< "30÷7のあまりの数は" << ans4 << "です。¥n";
    cout<< "(7+32)÷5は" << ans5 << "です。¥n";

    return 0;
}
```

2.

```
#include <iostream>
using namespace std;

int main()
{
    double height,width;

    cout << "三角形の高さを入力してください。¥n";
    cin >> height;
    cout << "三角形の底辺を入力してください。¥n";
    cin >> width;
    cout << "三角形の面積は" << height * width / 2 << "です。¥n";

    return 0;
}
```

3.

```
#include <iostream>
using namespace std;

int main()
{
    int sum=0, num=0;
```

```
    cout  <<  "科目1の点数を入力してください。¥n";
    cin >> num;
    sum += num;
    cout  <<  "科目2の点数を入力してください。¥n";
    cin >> num;
    sum += num;
    cout  <<  "科目3の点数を入力してください。¥n";
    cin >> num;
    sum += num;
    cout  <<  "科目4の点数を入力してください。¥n";
    cin >> num;
    sum += num;
    cout  <<  "科目5の点数を入力してください。¥n";
    cin >> num;
    sum += num;
    cout  <<  "5科目の合計点は"  <<  sum  <<  "点です。¥n";
    cout  <<  "5科目の平均点は"  <<  (double)sum/5  <<  "点です。¥n";

    return 0;
}
```

なお、第6章で学ぶ繰り返し文などを使うと、さらに簡単なコードを記述することができます。

Lesson 5 場合に応じた処理

1.

```
#include <iostream>
using namespace std;

int main()
{
    int res;

    cout  <<  "整数を入力してください。¥n";
    cin >> res;

    if((res % 2)==0)
        cout << res  <<  "は偶数です。¥n";
    else
        cout << res  <<  "は奇数です。¥n";

    return 0;
}
```

0でない整数はtrueと評価されることから、次のように記述することもできます。

```
#include <iostream>
using namespace std;

int main()
{
   int res;

   cout << "整数を入力してください。¥n";
   cin >> res;

   if(res % 2)
      cout << res << "は奇数です。¥n";
   else
      cout << res << "は偶数です。¥n";

   return 0;
}
```

2.

```
#include <iostream>
using namespace std;

int main()
{
   int num1, num2;

   cout << "2つの整数を入力してください。¥n";
   cin >> num1 >> num2;
   if (num1 < num2) {
      cout << num1 << "より" << num2 <<
         "のほうが大きい値です。¥n";
   }
   else if(num1 > num2) {
      cout << num2 << "より" << num1 <<
         "のほうが大きい値です。¥n";
   }
   else{
      cout << "2つの数は同じ値です。¥n";
   }

   return 0;
}
```

3.

```
#include <iostream>
using namespace std;

int main()
{
   int res;

   cout << "成績を入力してください。¥n";
   cin >> res;

   cout << "成績は" << res <<"です。";

   switch(res){
      case 1:
         cout << "もっとがんばりましょう。¥n";
         break;
      case 2:
         cout << "もう少しがんばりましょう。¥n";
         break;
      case 3:
         cout << "さらに上をめざしましょう。¥n";
         break;
      case 4:
         cout << "たいへんよくできました。¥n";
         break;
      case 5:
         cout << "たいへん優秀です。¥n";
         break;
   }

   return 0;
}
```

Lesson 6 何度も繰り返す

1.

```
#include <iostream>
using namespace std;

int main()
{
   cout << "1～10までの偶数を出力します。¥n";
   for(int i=1; i<=10; i++){
```

```
        if((i % 2) == 0)
            cout << i << '¥n';
    }
    return 0;
}
```

2.

```
#include <iostream>
using namespace std;

int main()
{
    int num, sum=0;
    cout << "テストの点数を入力してください。(0で終了)¥n";
    do{
        cin >> num;
        sum += num;
    }while(num);

    cout << "テストの合計点は" << sum << "点です。¥n";

    return 0;
}
```

3.

```
#include <iostream>
using namespace std;

int main()
{
    int s = 1;
    for(int i=0; i<5; i++){
        for(int j=0; j<s; j++){
            cout << '*';
        }
        cout << '¥n';
        s++;
    }

    return 0;
}
```

Lesson 7 関数

1.

```
#include <iostream>
using namespace std;

//square関数の宣言
int square(int x);

//square関数の呼び出し
int main()
{
   int num1;
   int sq1;

   cout << "整数を入力してください。¥n";
   cin >> num1;
   sq1 = square(num1);
   cout << num1 << "の2乗は" << sq1 << "です。¥n";

   return 0;
}

//square関数の定義
int square(int x)
{
   return x * x;
}
```

2.

```
#include <iostream>
using namespace std;

//square関数の宣言
int square(int x);
double square(double x);

//square関数の呼び出し
int main()
{
   int num1;
   int sq1;

   cout << "整数を入力してください。¥n";
   cin >> num1;
```

```
   sq1 = square(num1);
   cout << num1 << "の2乗は" << sq1 << "です。¥n";

   double num2;
   double sq2;
   cout << "小数を入力してください。¥n";
   cin >> num2;
   sq2 = square(num2);
   cout << num2 << "の2乗は" << sq2 << "です。¥n";

   return 0;
}

//square関数(int型)の定義
int square(int x)
{
   return x * x;
}

//square関数(double型)の定義
double square(double x)
{
   return x * x;
}
```

3.

```
#include <iostream>
using namespace std;

//square関数の定義
inline int square(int x){return x*x;}
inline double square(double x){return x*x;}

//square関数の呼び出し
int main()
{
   int num1;
   int sq1;
   cout << "整数を入力してください。¥n";
   cin >> num1;
   sq1 = square(num1);
   cout << num1 << "の2乗は" << sq1 << "です。¥n";

   double num2;
   double sq2;
```

```
    cout << "小数を入力してください。¥n";
    cin >> num2;
    sq2 = square(num2);
    cout << num2 << "の2乗は" << sq2 << "です。¥n";

    return 0;
}
```

4.

```
#include <iostream>
using namespace std;

//squaret関数テンプレートの定義
template <class T>
T squaret(T x)
{
    return x * x;
}

//squaret関数の呼び出し
int main()
{
    int num1;
    int sq1;
    cout << "整数を入力してください。¥n";
    cin >> num1;
    sq1 = squaret(num1);
    cout << num1 << "の2乗は" << sq1 << "です。¥n";

    double num2;
    double sq2;
    cout << "小数を入力してください。¥n";
    cin >> num2;
    sq2 = squaret(num2);
    cout << num2 << "の2乗は" << sq2 << "です。¥n";

    return 0;
}
```

Lesson 8 ポインタ

1. ① ×　② ○　③ ×

2.

```
#include <iostream>
using namespace std;

//add関数の宣言
void add(int* x1, int* x2, int a);

int main()
{
   int num1 = 0;
   int num2 = 0;
   int ad = 0;

   cout << "2科目分の点数を入力してください。¥n";
   cin >> num1 >> num2;
   cout << "加算する点数を入力してください。¥n";
   cin >> ad;
   add(&num1, &num2, ad);
   cout << ad << "点加算しましたので¥n";
   cout << "科目1は" << num1 << "点となりました。¥n";
   cout << "科目2は" << num2 << "点となりました。¥n";

   return 0;
}

//add関数の定義
void add(int* x1, int* x2, int a)
{
   *x1 += a;
   *x2 += a;
}
```

3.

```
#include <iostream>
using namespace std;

//add関数の宣言
void add(int& x1, int& x2, int a);

int main()
{
   int num1 = 0;
   int num2 = 0;
   int ad = 0;
```

```
    cout << "2科目分の点数を入力してください。¥n";
    cin >> num1 >> num2;
    cout <<"加算する点数を入力してください。¥n";
    cin >> ad;
    add(num1, num2, ad);
    cout << ad << "点加算しましたので¥n";
    cout << "科目1は" << num1 << "点となりました。¥n";
    cout << "科目2は" << num2 << "点となりました。¥n";

    return 0;
}

//add関数の定義
void add(int& x1, int& x2, int a)
{
    x1 += a;
    x2 += a;
}
```

Lesson 9 配列

1.

```
#include <iostream>
using namespace std;

//max関数の宣言
int max(int x[]);

int main()
{
    int test[5];
    cout << "テストの点数を入力してください。¥n";
    for(int i=0; i<5; i++){
        cin >> test[i];
    }
    int m = max(test);
    cout << "最高点は" << m << "点です。¥n";

    return 0;
}

//max関数の定義
int max(int x[])
{
```

```
    int m = x[0];
    for(int i=1; i<5; i++){
        if(m < x[i])
            m = x[i];
    }
    return m;
}
```

2.

```
#include <iostream>
using namespace std;

//length関数の宣言
int length(char* c);

int main()
{
    char str[100];
    cout << "文字列を入力してください。¥n";
    cin >> str;
    int ln = length(str);
    cout << "文字列の長さは" << ln << "です。¥n";

    return 0;
}

//length関数の定義
int length(char* c)
{
    int i = 0;
    while(c[i]){
        i++;
    }
    return i;
}
```

3.

```
#include <iostream>
using namespace std;

//count関数の宣言
int count(char str[], char c);

int main()
```

```
{
   char str[100];
   char ch;

   cout << "文字列を入力してください。¥n";
   cin >> str;
   cout << "文字列から探す文字を入力してください。¥n";
   cin >> ch;
   int c = count(str, ch);
   cout << str << "の中に" << ch << "は" << c <<
      "個あります。¥n";

   return 0;
}

//count関数の定義
int count(char str[], char ch)
{
   int i = 0;
   int c = 0;
   while(str[i]){
      if(str[i] == ch)
         c++;
      i++;
   }
   return c;
}
```

Lesson 10 大きなプログラムの作成

1. ① ○　② ○　③ ×　④ ×　⑤ ○

2. delete演算子を使ってメモリを解放します。

```
#include <iostream>
using namespace std;

int main()
{
   int* pA;
   pA = new int;
   *pA = 10;
   delete pA;
```

```
    return 0;
}
```

Lesson 11 いろいろな型

1.

```
#include <iostream>
using namespace std;

//構造体型Personの宣言
struct Person{
    int age;
    double weight;
    double height;
};

int main()
{
    Person ps[2];

    for(int i=0; i<2; i++){
        cout << "年齢を入力してください。¥n";
        cin >> ps[i].age;
        cout << "体重を入力してください。¥n";
        cin >> ps[i].weight;
        cout << "身長を入力してください。¥n";
        cin >> ps[i].height;
    }

    for(int j=0; j<2; j++){
        cout << "年齢" << ps[j].age << "体重" << ps[j].weight
            << "身長" << ps[j].height << "です。¥n";
    }
    return 0;
}
```

2.

```
#include <iostream>
using namespace std;

//構造体型Personの宣言
struct Person{
```

```
    int age;
    double weight;
    double height;
};

//aging関数の定義
void aging(Person* p)
{
    p->age++;
}

int main()
{
    Person ps;

    cout << "年齢を入力してください。¥n";
    cin >> ps.age;
    cout << "体重を入力してください。¥n";
    cin >> ps.weight;
    cout << "身長を入力してください。¥n";
    cin >> ps.height;

    cout << "年齢" << ps.age << "体重" << ps.weight << "身長"
        << ps.height <<"です。¥n";

    aging(&ps);
    cout << "1年経過しました。¥n";

    cout << "年齢" << ps.age << "体重" << ps.weight << "身長"
        << ps.height <<"です。¥n";

    return 0;
}
```

Lesson 12 クラスの基本

1. ① ×　② ○　③ ×　④ ○　⑤ ○

2および3

```
#include <iostream>
using namespace std;

//Pointクラスの宣言
```

```
class Point{
   private:
      int x;
      int y;
   public:
      void setX(int a);
      void setY(int b);
      int getX(){return x;}
      int getY(){return y;}
};

//Pointクラスメンバ関数の定義
void Point::setX(int a)
{
   if(0 <= a && a <= 10)
      x = a;
   else
      x = 0;
}
void Point::setY(int b)
{
   if(0 <= b && b <= 10)
      y = b;
   else
      y = 0;
}

int main()
{
   Point p;
   int x, y;

   cout << "X座標を入力してください。¥n";
   cin >> x;
   cout << "Y座標を入力してください。¥n";
   cin >> y;

   p.setX(x);
   p.setY(y);

   cout << "座標は(" << p.getX() << "," << p.getY() <<
      ")です。¥n";

   return 0;
}
```

Lesson 13 クラスの機能

1. ① × ② ○ ③ ×

2. ① × ② × ③ ×

Lesson 14 新しいクラス

1. ① × ② × ③ ○

2. ① × ② ○ ③ ○

3. main()関数内のdrv.showBs();の呼び出しがあいまいです。

Lesson 15 クラスに関する高度なトピック

1. ① ○ ② ○ ③ ×

2. ① × ② ○ ③ ×

3. 4.

```
#include <iostream>
using namespace std;

//Pointクラスの宣言
class Point{
   private:
      int x;
      int y;
   public:
      Point(int a=0, int b=0){x=a; y=b;}
      void show(){cout<< " x :" << x <<" y :" << y <<'¥n';}
      Point operator++();
      Point operator++(int d);
      Point operator--();
      Point operator--(int d);
      friend Point operator+(Point p1, Point p2);
      friend Point operator+(Point p, int a);
      friend Point operator+(int a, Point p);
      friend Point operator-(Point p1, Point p2);
```

```
        friend Point operator-(Point p, int a);
        friend Point operator-(int a, Point p);
};

//Pointクラスメンバ関数の定義
Point Point::operator++()
{
    x++;
    y++;
    return *this;
}
Point Point::operator++(int d)
{
    Point p = *this;
    x++;
    y++;
    return p;
}
Point Point::operator--()
{
    x--;
    y--;
    return *this;
}
Point Point::operator--(int d)
{
    Point p = *this;
    x--;
    y--;
    return p;
}

//フレンド関数の定義
Point operator+(Point p1, Point p2)
{
    Point tmp;
    tmp.x = p1.x + p2.x;
    tmp.y = p1.y + p2.y;
    return tmp;
}
Point operator+(Point p, int a)
{
    Point tmp;
    tmp.x = p.x + a;
    tmp.y = p.y + a;
    return tmp;
```

```
}
Point operator+(int a, Point p)
{
   Point tmp;
   tmp.x = a + p.x;
   tmp.y = a + p.y;
   return tmp;
}
Point operator-(Point p1, Point p2)
{
   Point tmp;
   tmp.x = p1.x - p2.x;
   tmp.y = p1.y - p2.y;
   return tmp;
}

Point operator-(Point p, int a)
{
   Point tmp;
   tmp.x = p.x - a;
   tmp.y = p.y - a;
   return tmp;
}
Point operator-(int a, Point p)
{
   Point tmp;
   tmp.x = a - p.x;
   tmp.y = a - p.y;
   return tmp;
}

int main()
{
   Point p1(20,10);
   Point p2(1,2);
   p1 = p1-p2;
   p1--;
   p1 = p1-3;
   p2 = 3-p2;

   p1.show();
   p2.show();

   return 0;
}
```

Lesson 16 ファイルの入出力

1.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   for (int i=0; i<=5; i++){
      for (int j=1; j<=5; j++){
         cout.width(3);
         cout.fill('-');
         cout << i*5+j;
        }
    cout << '¥n';
   }
   return 0;
}
```

2.

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

int main(int argc, char* argv[])
{
   if(argc != 2){
      cout << "パラメータの数が違います。¥n";
      return 1;
   }

   ifstream fin(argv[1]);
   if(!fin){
      cout << "ファイルをオープンできませんでした。¥n";
      return 1;
   }

   const int num = 8;
   int test[num];
   for(int i=0; i<num; i++){
      fin >> test[i];
   }
   int max = test[0];
```

```
    int min = test[0];
    for(int j=0; j<num; j++){
       if(max < test[j])
          max = test[j];
       if(min > test[j])
          min = test[j];
       cout << "No." << j+1 << setw(5) << test[j] << '¥n';
    }

    cout << "最高点は"<< max <<"です。¥n";
    cout << "最低点は"<< min <<"です。¥n";

    fin.close();

    return 0;
}
```

Index

記号

数字

B

C

D

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

ふ

へ

ほ

ま

み

め

も

ゆ

よ

ら

り

る

れ

ろ

●著者略歴

高橋 麻奈

1971年、東京生まれ。東京大学経済学部卒業。出版社に勤務の後、テクニカルライターとして独立。

主な著作に、『やさしいC』『やさしいC++』『やさしいJava』『やさしいJava 活用編』『やさしいXML』(以上、ソフトバンク パブリッシング)、『入門SQL』(ソフトバンク パブリッシング、共著)、『図解でわかるXMLのすべて』『ここからはじめるデータベース』『ここからはじめるC言語』(以上、日本実業出版社)、『それゆけ! ビジュアルベーシック』(日経BP社) などがある。

本書のサポートページ: **http://www.bekkoame.ne.jp/~manachan/yasac.html**

やさしいC++(シープラスプラス) 第2版

2000年 3月30日 初版発行
2003年 9月30日 第2版 第1刷発行
2004年 1月30日 第2版 第2刷発行

著　者 高橋(たかはし) 麻奈(まな)
編集協力 風工舎

発行者 稲葉 俊夫
発行所 ソフトバンク パブリッシング株式会社
〒107-0052 東京都港区赤坂4-13-13
販　売 03-5549-1200
編　集 03-5549-1145

印　刷 文唱堂印刷株式会社

装　丁 ハヤカワデザイン (ikuo@mvj.biglobe.ne.jp)
早川いくを／高瀬はるか

落丁本、乱丁本は小社販売局にてお取り替えします。
定価はカバーに記載されております。

Printed in Japan　ISBN4-7973-2476-7

SOFT BANK Publishing

ISBN4-7973-2476-7

C0055 ¥2600E

9784797324761

定価 本体2,600円 +税

1920055026000

やさしいC++ 第2版

高橋麻奈 著

SOFT BANK Publishing